FORTRAN D PARALLELIZING COMPILER: RESTRUCTURING PHASE

A Thesis Submitted
In Partial Fulfilment of the Requirements
for the Degree of
MASTER OF TECHNOLOGY

by
ANANDA R

to the

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY
KANPUR

February, 1993

CERTIFICATE

This is to certify that the work contained in the thesis titled **FORTRAN D PARALLELIZING COMPILER: RESTRUCTURING PHASE** by **ANANDA R**, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Sanjeev Kumar

Dr. Sanjeev Kumar Aggarwal
Assistant Professor,
Department of Computer
Science and Engineering
IIT, Kanpur

Acknowledgements

Many persons have not only helped but shown me the way during the period of my M.Tech thesis work, but even amongst these, Dr. Sanjeev Kumar Aggarwal occupies a special place in my mind. As an instructor for one of the courses in my first semester, he drew me into this exciting field of parallelizing compilers. And as my thesis supervisor he has not only given a sense of direction to my work, but has been a tremendous source of motivation. The choicest of epithets would not suffice to express my gratitude for him.

Rajeev and Gautam's active participation in the project and our weekly meetings helped me to analyze problems from a different perspective. Thanks to them, I have picked up the threads of team-work. Murali, Kumar and Shankara were instrumental in helping me break down nervous energy before the defense. Sarkar's suggestions on the style of report writing have been most useful.

A number of people have stood by me during the past year and a half. I would particularly like to mention my classmates, who have been an absolutely wonderful bunch and the members of the Kannada Sangha, who made me feel at home even in Kanpur and made a strong impact in moulding my personality. The faculty members of our department were ever so cooperative during my course work.

Last but not the least, hats off to Leslie Lamport, the author of LaTeX, who has been the biggest reason why (*I* believe) this thesis looks presentable. And the lethargy of the Cul-secy in getting the Hall 5 stereo system repaired, thanks to which, the music room's silence was like a poultice that healed the hard blows of sound rendered ceaselessly by those blaring loudspeakers in nearby Nankari!

Ananda R
Kanpur
Feb 09, 1993.

To all those who have
helped me get this far—
Foremost among them
My parents and teachers.

Abstract

In the last decade, multiprocessor architectures have been developed that have the potential to bypass the limits on speed imposed by sequential machines. One of the problems encountered in effectively tapping the parallelism provided by the machine is the difficulty in programming such systems. Unlike programs written for traditional von Neumann architectures, parallel programming involves a number of bookkeeping chores. The resistance faced in adapting to newer programming techniques and the enormous amount of investment made in the existing software calls for an approach that automatically restructures sequential programs into the desired parallel program. FRAMES is a restructuring aid for programs written in FORTRAN D/FORTRAN 77 that automatically detects the parallelism inherent in a sequential program. In this thesis, we discuss the design and implementation issues that went into the development of FRAMES.

Contents

Chapter 1

Introduction

The successive generations of computers have shared a common goal in the urge to facilitate faster computation. The last decade has seen an unmistakable shift towards parallel processing systems in scientific applications to bypass the limits imposed by sequential computing. Multiprocessing, in particular, best embodies this philosophy of putting more processors at work to improve the turnaround time of a program.

Despite the promise of high performance held out by multiprocessors, realizing it is seldom possible. This is primarily because progress made in the languages which support constructs to effectively express parallelism has been slow. A programmer in such a language invariably has to keep track of a number of housekeeping tasks which would merely obscure the application at hand. The (hardly insignificant) investment that has been made in software for systems prevalent before the advent of parallel processing (the fate of *dusty decks*, to quote Polychronopoulos [13]) cannot be discounted. What is needed is a mechanism to *automate* the translation of programs written in languages used so far into a language used for machines based on newer architectures. Such a *parallelizing* (*restructuring*) compiler would be a boon to a programmer as it buffers him from the changes in underlying machine architecture and the languages

used to program it[1].

Parallelizing compilers such as PTRAN, PFC and PARAFRASE, which restructure a program in a given sequential language, have been developed before. Like the code generated by a sequential compiler which is very often worse than that produced by a good assembly language programmer, the restructured code too may not be fully optimized. Given an opportunity, an intelligent programmer may improve the code further. Since the restructured code may be represented in a high-level language form, an attractive proposition is to present the restructured program to the user (for more re-restructuring?) whose intuitive understanding of the application at hand may enable him to perform/give directives for further parallelization. This thesis work is part of an ongoing project to build such a parallelizing compiler called FRAMES (**F**ortran D **R**estructuring **A**id for an **MES**) for a type of multiprocessor architecture called MES (described in section 1.2 below).

1.1 Structure of the parallelizing compiler

We describe below the structure of our parallelizing compiler, that is modular in nature and in which the different phases can be conveniently represented by blocks [11]. A block diagram depicting the various phases is shown in Figure 1.1.

Phase 1. The front end, as in any other compiler, converts the input program to a suitable intermediate representation. Besides parsing, global data flow analysis and machine independent optimizations are done in this phase.

Phase 2. The issue of restructuring involves analysis of the constructs in the program and applying the appropriate transformations to exploit potential parallelism in the program. Since loops are the main sources of parallelism

[1] "Bless thee, Bottom! bless thee! thou art translated", *A Mid-summer Night's Dream*

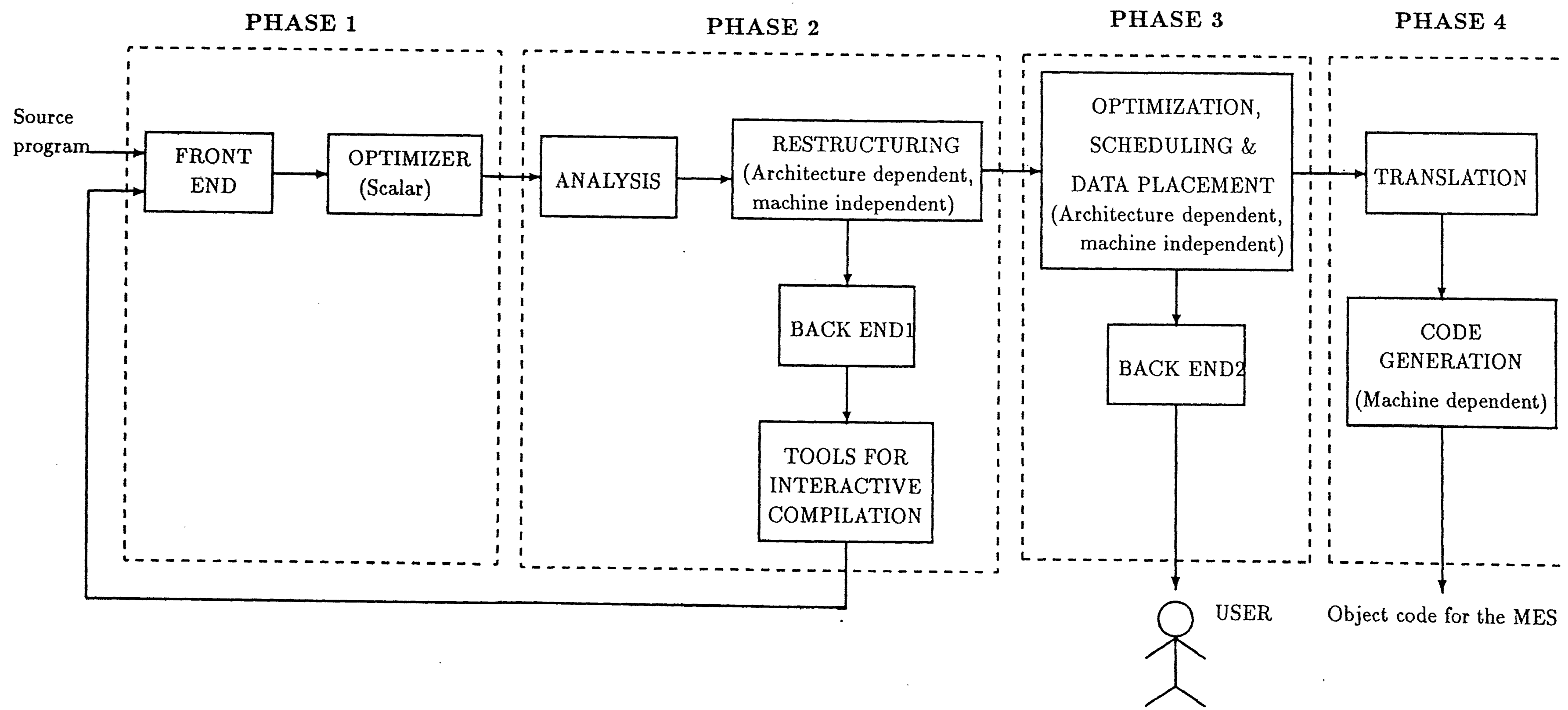


Figure 1.1: The structure of the compiler

(programs spend most of their time iterating over a portion of code), they come under close scrutiny. A *sine quâ non* for parallelization of a loop is the resolution of data dependences, to ensure that the restructured program is never at semantic variance with the original program.

Phase 3. This phase schedules the restructured program for execution on a multiprocessor system. By scheduling, we mean the assignment of iterations/tasks to processors. A task is a set/block of statements that are executed on any processor. Architecture dependent (but machine independent) optimizations, data placement (deciding where and how to store data when it is accessed by multiple processors) need to be done before the machine code is generated.

Phase 4. The final phase is completely machine dependent and generates the machine code for the restructured and scheduled program.

The hallmark of this structure is the facility to present the restructured program to the user for detecting more parallelism. Notice that BACK END1 in Phase 2 translates the intermediate representation to a high-level program. For reasons more than one, Fortran D[5] has been chosen as the high-level language for representing this parallel program—besides the parallel constructs offered, the language also supports statements that aid proper data placement. Moreover it is a superset of Fortran 77, the most extensively used language for scientific applications.

Presenting the restructured program to the user gives him an opportunity to give directives to the compiler for enhanced detection of parallelism. Phase 3 is entered only when the user is convinced that there can be little benefit gained by any more restructuring.

Equally significant, the presence of BACK END2 permits the user to view the program after data placement. For a user who is familiar with data distribution specifications, the presence of BACK END2 permits him to view the program and study the effect of a data distribution command.

In effect, this structure can be viewed as either of the following:

1. As a Fortran D-Fortran D parallelism detector (Phase 1 + Phase 2 with the feedback). Since Fortran 77 is a subset of Fortran D, we could also view this part as a Fortran 77-Fortran D restructurer.

2. As a single black box that maps sequential programs to the underlying multiprocessor architecture (the entire compiler).

This thesis deals with the problem of restructuring in Phase 2. We shall address the nuances of restructuring with the objective of decreasing the overall run time of the parallel program without any substantial increase in compile-time.

1.2 The machine model and the language assumed

FRAMES is being developed for a multiprocessor architecture that has a global shared memory accessible by all processors in addition to some limited local memory that is private to each processor. Each processor itself is scalar; thus such a model can be categorized under Kuck's taxonomy as an **MES** (Multiple Execution array of Scalar processors) architecture [13]. The restructurer is insulated from the details of the underlying machine model by Phase 3. A popular theory currently held by many researchers is that it is not so much the restructuring techniques but the inefficiencies in data distribution that are impeding higher throughput in a multiprocessor. Fortran D [5] lays particular emphasis on this aspect and supports a rich variety of data distribution (placement)

specifications. Appendix A gives a brief overview of Fortran D as relevant to this thesis. For a more detailed description refer [5]. The restructurer accepts Fortran D statements and generates the parallel equivalent of all serial loops where legal.

1.3 Overview of the report

The motivation for a restructuring compiler apart, other issues have been dealt with in the ensuing chapters.

In **Chapter 2**, we consider the problems in data dependence analysis and survey the techniques that have been developed to speedily resolve them.

Chapter 3 is a survey of well established restructuring methods which utilize the results of data dependence analysis to extract parallelism from the program.

A crucial aspect is the order in which the different restructuring heuristics are applied. The algorithm used for generating the parallel program in Fortran D is covered in **Chapter 4**. It is also necessary to determine if application of a technique ever results in incorrect semantics or generation of redundant code. Feasibility tests are discussed for this.

All implementation specific details have been confined to **Chapter 5** while **Chapter 6** presents the results hitherto obtained in the existing prototype of FRAMES, with suggestions to improve upon them.

Appendix A gives a summary of Fortran D. We have listed the grammar accepted by our parser for Fortran D in **Appendix B**.

With this bird's eye-view of the thesis, we now zero in on each of the topics mentioned.

Chapter 2

Data Dependence Analysis Techniques : A Survey

In this chapter, we survey a class of techniques used in data dependence analysis. This analysis has to be done to identify the statements in a loop which are free from cyclic dependences i.e., those statements which can be executed in a parallel loop. Clearly, an effective scheme is needed to detect the presence/absence of dependences, so that the restructuring phase can use its results to tap the maximum possible parallelism from the given sequential program.

2.1 Definitions and notations

A statement T depends on a statement S if there exist two instances of S and T, say S' and T' respectively, such that

- both S' and T' reference a memory location M
- S' is executed before T' in the serial execution of the program
- at least one of the references, by S' or T', is a write to M
- M is not written in between the accesses to it by S' and T'.

We classify dependences into the following categories.

- A *flow-dependence* is said to exist from S to T if S computes and writes data that can subsequently be read by the statement T.
- An *anti-dependence* exists from S to T if S reads data from a location into that T can subsequently write.
- An *output-dependence* exists from S to T if S writes into a location which is subsequently written into by T also.

In each of the above cases, interchanging the order of the two statements S and T would result in altering the semantics of the original program. More formally, given a loop nest

```
DO I1 = 1, N1
   DO I2 = 1, N2
      ⋮
      DO In = 1, Nn
S:          X(f(I1,I2,...,In)) = ···
T:          ··· = X(g(I1,I2,...,In))
      ENDDO
      ⋮
   ENDDO
ENDDO
```

(2.1)

A dependence exists from S to T if

$$f(i_1, i_2, \ldots, i_n) = g(j_1, j_2, \ldots, j_n) \tag{2.2}$$

where i_k, j_k are instances of $I_k, 1 \le k \le n$

Direction vector. When a dependence exists between two statements, it is very useful to summarize the constraints that are imposed on the values that the instances i_k and j_k can take. Accordingly, we associate a *direction* with each loop index variable:

- "<" indicates that $i_k < j_k$ i.e., the dependence exists from an iteration to a subsequent iteration
- "=" indicates that $i_k = j_k$ i.e., the dependence exists in the same iteration
- ">" indicates that $i_k > j_k$ i.e., the dependence holds from one iteration to an earlier iteration
- "*" indicates that no constraints are imposed on the pair of iteration values, i_k and j_k.

Clearly, "*" is a combination of "<", "=" and ">".

By combining the directions induced by each loop on a dependence, we can construct the *direction vector* for the dependence.

Distance vector. The direction vector gives us an idea of the order in which the different instances of iterations access a variable. Although this information is usually sufficient for most restructuring techniques, for a thorough analysis of the program, it is necessary to know the difference $(j_k - i_k)$ between the two iteration instances, or the *distance* of the dependence corresponding to that direction. A *distance vector* comprises the distance of the dependence induced by each loop index variable.

Level and depth of a dependence. Given a dependence with a direction vector of the form $(=, =, \ldots, <, *, \ldots, *)$ with the first "<" being the lth component of the direction vector, the dependence is said to occur at *level* l. If the variables cause a dependence at a level $l \geq d$, then the dependence exists at a *depth* d.

2.2 Inexact tests

The general form of equation (2.1) for an array reference that has m subscripts is

```
    DO I_1 = 1, N_1
       DO I_2 = 1, N_2
          ⋮
          DO I_n = 1, N_n
S:           A(f_1(I_1,I_2,...,I_n),...,f_m(I_1,I_2,...,I_n)) = ···
T:           ··· = A(g_1(I_1,I_2,...,I_n),...,g_m(I_1,I_2,...,I_n))
          ENDDO
          ⋮
       ENDDO
    ENDDO
```

Clearly for a dependence to exist, we must have

$$\begin{aligned} f_1(i_1, i_2, \ldots, i_n) &= g_1(j_1, j_2, \ldots, j_n) \\ f_2(i_1, i_2, \ldots, i_n) &= g_2(j_1, j_2, \ldots, j_n) \\ &\vdots \\ f_m(i_1, i_2, \ldots, i_n) &= g_m(j_1, j_2, \ldots, j_n) \end{aligned} \tag{2.3}$$

where every i_k and j_k are two instances of the corresponding loop index I_k.

Not all solutions to the system of equations (2.3) are of interest to us since the loop index variables take on only integer values. Hence, we need to detect the presence of only integer solutions. Further, the integer solution that we seek should also lie within the region that is described by the bounds of the loop index variables. This is because the solution should represent the iterations between which a dependence exists.

Unfortunately, solving such a system of equations is an integer programming problem which would be fairly cost prohibitive to apply per se. Thus with an

eye on reducing the compile time taken to restructure the program, it is imperative that we perform a quick test whose inaccuracies are on the conservative side. Further, we prefer a subscript-by-subscript dependence checking method to simultaneous solution of the system of equations. This approach is not as pessimistic as it may sound—results have shown that the inexact tests can handle a large class of real-world problems with little/no loss in accuracy. In a later section, we outline some heuristics to reduce the inaccuracies caused by inexact tests.

The inexact tests do *not* detect the exact instances of a loop index variable at which the dependence exists; rather they address the less complex question, "does a dependence exist?".

FRAMES currently applies only some of the inexact tests for checking the existence of a dependence between two array references. As a result, certain restructuring techniques that need to know the distance of a dependence have not been implemented in this framework. Hybrid tests exist that can calculate the distance of a dependence.

Linear diophantine equations.

A realistic assumption that is made is in the degree of the equation (2.1). An overwhelming majority of array references in scientific code are linear functions of the loop index variables, and it is tempting not to overlook this observation. Equations of this type are referred to as *linear diophantine equations* and the general form of this is

$$a_1x_1 + a_2x_2 + \ldots + a_nx_n = c \tag{2.4}$$

where $(x_1, x_2, \ldots, x_n)$ is the solution sought.

We shall consider (2.4) as the equation to be solved in each of the inexact tests that we discuss.

A few remarks on these tests now would be in order.

- **Existence** of a solution indicates the (*possible*) *presence* of a dependence. We stress the word possible since inaccuracies in the tests can cause them to report erroneous results, but on the conservative side.
- **Failure** of a test means that the test *failed to detect the absence of a dependence*. In other words, if the test determines the existence of a solution that does not actually exist, then the test has failed. This too is caused by inaccuracies in the testing procedure.

These terms can be better understood in the examples given later in this chapter.

2.3 The GCD test

One of the earliest dependence tests realized was due to an important result from number theory regarding linear diophantine equations. It makes use of the concept of the *greatest common divisor* (gcd) of two integers. We state the theorem but omit the proof due to pressures of space.

Theorem 2.1 The linear diophantine equation

$$a_1x_1 + a_2x_2 + \ldots + a_nx_n = c$$

has a solution in integer space iff $\gcd(a_1, a_2, \ldots, a_n)$ divides c.

For proof, the reader is referred to [3]. ∎

It is seen that the GCD test attempts to detect a solution in the entire field of integers $\mathcal{Z}$, without taking the bounds of the loop index variables into account. As a result, the test fails when it hits on a solution that is outside our region of interest. It also implies that we cannot detect the direction of a dependence

using the GCD test because the test does not take advantage of the range of each loop index variable.

More disconcerting is a property of the gcd of n non-negative integers, viz.

$$\gcd(a_1, a_2, \ldots, a_n) \leq \min(|a_1|, |a_2|, \ldots, |a_n|)$$

Clearly, even a single coefficient whose absolute value is unity can hold the test to ransom as it reduces the gcd to 1.

Like other inexact tests, this test can be used as a necessary condition for the existence of a dependence. A very useful feature of this test is that it forms the basis of an exact test that finds a general parametric solution using matrix methods ([3]), called the *Extended GCD test.*

The Extended GCD Test

The extended GCD test described below is exact *ignoring bounds*. For multi-dimensional array references, a matrix method may be employed to get a parametric solution to the system of equations that result from multiple diophantine equations, one in each array dimension.

Let $\mathbf{x}\mathbf{A} = c$ be the system of equations. Banerjee showed that we can factor this into

- a **unimodular integer matrix U**, which has the property that the determinant of **U** is ± 1.
- an **echelon matrix D**, which has the property that if the first non-zero element in row i is in column j, then the first non-zero element in row $i+1$ is in column $k > j$.
- **D** and **U** satisfy the relation $\mathbf{UA} = \mathbf{D}$.

This factoring process is very similar to Gauss-Jordan reduction. Now, the system $\mathbf{x}\mathbf{A} = c$ has an integer solution $\mathbf{x}$ iff there exists an integer vector $\mathbf{t}$ such that $\mathbf{t}\mathbf{D} = c$. As $\mathbf{D}$ is echelon, such a vector if it exists can easily be obtained by back substitution. Absence of such a $\mathbf{t}$ indicates the absence of a solution to the system of equations. Finally, the solution to the system of equations can be obtained by the relation $\mathbf{x} = \mathbf{t}\mathbf{U}$.

2.4 Banerjee's test

Banerjee's test is motivated by the Lagrange's Intermediate Value Theorem of advanced calculus. This theorem brings out an important property of continuous functions.

Theorem 2.2 Let f be a continuous real-valued function on $\mathbf{R}^n$. Let b, B denote any two values of f on a region $\Re \subset \mathbf{R}^n$ and let $b \leq c \leq B$. Then the equation

$$f(\mathbf{x}) = c$$

has a solution $\mathbf{x}$ in $\Re$. ∎

Thus, if we can evaluate the minimum and maximum values of the function, b and B, and verify if c lies in between these two extremities, then there exists a real solution point $\mathbf{x}$ in $\Re$ which would satisfy $f(\mathbf{x}) = c$. Notice that what we actually seek is an *integer* solution to the equation $f(\mathbf{x}) = c$, whereas this method only guarantees a *real* solution point $\mathbf{x}$ that lies within the bounds—this is the inaccuracy introduced by Banerjee's test.

Notation.

$$\begin{aligned} a^+ &= a \text{ if } a > 0 \\ &= 0 \text{ otherwise} \end{aligned}$$

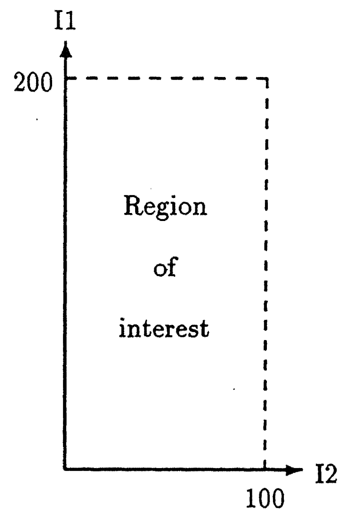
$$
\begin{aligned}
a^- &= -a \text{ if } a < 0 \\
&= 0 \text{ otherwise}
\end{aligned}
$$

2.4.1 Banerjee's test for rectangular spaces

Since the evaluation of the bounds of a function f depends on the type of the region over which f is defined, it is convenient to treat each region distinctly.

A *rectangular region* $\Re$ is one in which the variables describing the region are all independent of one another and have fixed bounds. For instance, in the loop

```
DO I1 = 1, 100
   DO I2 = 1, 200
       :
   ENDDO
ENDDO
```



any function f(I1, I2) would take values in the rectangular region,

$$\{1 \leq \mathtt{I1} \leq 100;\ 1 \leq \mathtt{I2} \leq 200\}$$

Theorem 2.3 Consider the linear diophantine equation

$$f(x) = a_1x_1 + a_2x_2 + \cdots + a_nx_n = c$$

with $p_k \leq x_k \leq q_k$. Then,

$$f_{\text{min}} = \sum_{k}^{n}(a_k^+ p_k - a_k^- q_k)$$

$$f_{\max} = \sum_{k}^{n}(a_k^+ q_k - a_k^- p_k)$$

A solution exists if $f_{\min} \leq c \leq f_{\max}$. ■

Example 2.1 Consider the following program segment

```
DO I=1, 2
   A(3*I) = ...
   ... = A(2*I+5)
ENDDO
```

Then, $3i - 2j = 5$ is the linear diophantine equation to be solved. The bounds for i and j are

$$1 \leq i \leq 3 \text{ and } 1 \leq j \leq 3$$

By GCD test, $\gcd(3, -2) = 1$.

By Banerjee's inequality,

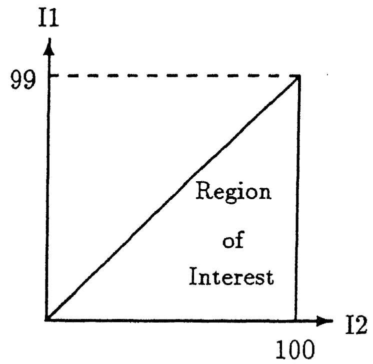
$$\begin{aligned} f_{\min} &= 3(1) - 2(2) &= -1 \\ f_{\max} &= 3(2) - 2(1) &= 4 \end{aligned}$$

As 5 does not lie within the bounds [-1, 4], no real solution and hence no integer solution can exist within these limits. Notice that the GCD test has failed in this case. ■

2.4.2 Trapezoidal regions

In a trapezoidal region, the variables describing the region are themselves dependent on the values taken by the other variables. For example, the loops in the selection sort routine are as shown below.

```
DO I1 = 1, 99
   DO I2 = I1+1, 100
       :
   ENDDO
ENDDO
```



the values that I2 can take are dependent on I1. In general, for the function

$$f(\mathbf{x}) = c,$$

the bounds on $x_1, x_2, \ldots, x_n$ are

$$p_{10} \le x_1 \le q_{10}$$

$$p_{20} + p_{21}x_1 \le x_2 \le q_{20} + q_{21}x_1$$

$$\vdots$$

$$p_{n0} + p_{n1}x_1 + \cdots + p_{n,n-1}x_{n,n-1} \le x_n \le q_{n0} + q_{n1}x_1 + \cdots + p_{n,n-1}x_{n,n-1}$$

Algorithm 2.1 shows how to compute the lower and upper bounds for a trapezoidal region.

Comments on Banerjee's test.

- Since Banerjee's test actively considers the bounds of the loop index variables, intuitively it should perform better than the GCD test. This also implies that loop bounds need to be known at compile-time or at least an estimate of the bounds should be available.

- Since Banerjee's test searches for a real solution, which is a superset of the integer points in the region, Banerjee's test fails when there is a real but no integer solution to the linear diophantine equation. Thus, any solution

Algorithm 2.1 (Banerjee [3])

begin
1. $b_{\text{low}} = 0$; $b_{\text{up}} = 0$
   $(d_1, d_2, \ldots, d_k) \leftarrow (a_1, a_2, \ldots, a_k)$
   $(e_1, e_2, \ldots, e_k) \leftarrow (a_1, a_2, \ldots, a_k)$
2. **for** $k = n$ **downto** 1 **do**
   begin
   $b_{\text{low}} = b_{\text{low}} + d_k^+ p_{k0} - d_k^- q_{k0}$
   $b_{\text{up}} = b_{\text{up}} + e_k^+ p_{k0} - e_k^- q_{k0}$
   if $(k > 1)$ **then**
   begin
   $(d_1, d_2, \ldots, d_{k-1}) \leftarrow (d_1 + d_k^+ p_{k1} - d_k^- q_{k1},$
   $d_2 + d_k^+ p_{k2} - d_k^- q_{k2}, \ldots, d_{k-1} + d_k^+ p_{k,k-1} - d_k^- q_{k,k-1})$
   $(e_1, e_2, \ldots, e_{k-1}) \leftarrow (e_1 + e_k^+ p_{k1} - e_k^- q_{k1},$
   $e_2 + e_k^+ p_{k2} - e_k^- q_{k2}, \ldots, e_{k-1} + e_k^+ p_{k,k-1} - e_k^- q_{k,k-1})$
   end
   end
3. $f_{\min} = b_{\text{low}}$
   $f_{\max} = b_{\text{up}}$

end

that is claimed to have been detected by Banerjee's test should be treated with suspicion. On the other hand, absence of a real solution precludes the possibility of any integer solution as well, clearly indicating that errors are on the conservative side.

- Wolfe has extended Banerjee's test to handle arbitrary direction vectors (see [3] or [19]).

2.5 The I test

In view of the inaccuracies that are inherent in Banerjee's and GCD tests, it is conventional to apply both these tests in the determination of a dependence between two array references. Despite this, the approach fails when there is a

real, non-integer solution within the bounds and an integer solution outside the bounds. The I test [8] attempts to integrate these two methods and alleviate some of the discrepancies.

To begin with, we note that the Banerjee inequality given in Theorem 2.2 can be rewritten as

$$\sum_k (a_k^+ p_k - a_k^- q_k) \leq c \leq \sum_k (a_k^+ q_k - a_k^- p_k)$$

Thus

$$0 \leq c - \sum_k (a_k^+ p_k - a_k^- q_k)$$

and

$$c - \sum_k (a_k^+ q_k - a_k^- p_k) \leq 0$$

Hence a solution is deemed to exist if 0 lies within these bounds i.e.,

$$\text{if } 0 \in [\underbrace{c - \sum_k (a_k^+ q_k - a_k^- p_k)}_{S_1},\ \underbrace{c - \sum_k (a_k^+ p_k - a_k^- q_k)}_{S_2}]$$

A possible improvement to Banerjee's test can be obtained if we desist from computing the entire sums S_1 and S_2. Rather, we compute these sums for some selected a_is and interleave computation of these partial sums with the GCD test. These ideas are more formally stated below.

Notation. The interval equation

$$f(x) = a_1 x_1 + a_2 x_2 + \cdots + a_n x_n = [L, U] \tag{2.5}$$

is a shorthand for the set of equations,

$$\begin{aligned}
a_1 x_1 + a_2 x_2 + \cdots + a_n x_n &= L \\
a_1 x_1 + a_2 x_2 + \cdots + a_n x_n &= L + 1 \\
&\vdots \\
a_1 x_1 + a_2 x_2 + \cdots + a_n x_n &= U
\end{aligned} \tag{2.6}$$

The equation (2.5) is solvable iff *at least one of the equations* in (2.6) is solvable i.e., $\gcd(a_1, a_2, \cdots, a_n)$ should divide *some* integer in $[L, U]$.

It is at times useful to divide the entire interval equation by $\gcd(a_1, a_2, \ldots, a_n) = d$. Then

$$(a_1/d)x_1 + (a_2/d)x_2 + \ldots + (a_n/d)x_n = [\lceil L/d \rceil, \lceil U/d \rceil]$$

is equivalent to (2.5).

Theorem 2.4 Let $a_1, a_2, \ldots, a_n$, L and U be integers. For each k, $1 \le k \le n-1$, let each of p_k and q_k be either an integer with $p_k \le q_k$ or an unknown ("*"). Let $p_n \le q_n$. If $|a_n| \le U - L + 1$, then the interval equation

$$f(x) = a_1x_1 + a_2x_2 + \cdots + a_nx_n = [L, U]$$

is integer solvable in the bounds for $I_1, I_2, \ldots, I_n$ iff the interval equation

$$f(x) = a_1x_1 + a_2x_2 + \cdots + a_{n-1}x_{n-1} = [L - (a_n^+ q_n - a_n^- p_n), U - (a_n^+ p_n - a_n^- q_n)]$$

is integer solvable in the bounds for $I_1, I_2, \ldots, I_{n-1}$.

Proof. See Kong *et al.* [8] ■

The I test however has its limitations. It cannot be extended to trapezoidal regions on the lines of Algorithm 2.1 since the bounds of a variable are not always known at the time of transferring it to the RHS.

2.6 Other tests

Although we have concentrated on inexact tests so far, certain exact tests have been developed that are more useful, albeit at a certain cost. We briefly outline two such methods here.

Algorithm 2.2

coeff $= \{a_1, a_2, \ldots, a_n\}$
$L = U = a_0$
while ($\exists$ an a_i such that $|a_i| \leq U - L + 1$ and p_i, q_i are both known) {
    $L = L - (a_i^+ q_i - a_i^- p_i)$
    $U = U - (a_i^+ p_i - a_i^- q_i)$
    coeff $=$ coeff $- \{a_i\}$
    if (coeff $= \emptyset$) {
        if ($L \leq 0 \leq U$) {
            print(dependence exists);
            return;
        }
        else {
            print(no dependence exists);
            return;
        }
    }
    $d = \gcd(a_i)\ \forall a_i \in$ coeff
    if d does not divide an integer in $[L, U]$ {
        print(no dependence exists);
        return;
    }
    if ($d \neq 1$) {
        $a_i = a_i/d\ \forall a_i$
        $L = \lceil L/d \rceil$
        $U = \lfloor U/d \rfloor$
    }
    else {
        print(dependence may exist);
        return;
    }
}

2.6.1 Maydan's approach

Maydan *et al.* [10] take the view that employing inexact methods sacrifices potential parallelism. Rather, applying a chain of algorithms for data dependence analysis, each of which are exact for *some* special case inputs, could lead to much better results. More importantly, a *memoization technique* is also introduced in their scheme that remembers the results of previous applications of the tests.

The exact tests used in their scheme are summarized here.

1. **Banerjee's extended GCD test**, which is used as a preprocessing step for other tests. Although this test addresses the simpler problem of finding an integer solution without taking any bounds into consideration, the application of this test is primarily used to make a change of variables (refer section 2.3, page 13), thus simplifying the original problem.

2. **Single variable per constraint test.** If the solution to the generalized GCD test has at most one free variable, then each constraint becomes an upper or lower bound for the free variable. As each constraint is examined, the tightest bounds for each variable are kept track of. Finally, after all constraints have been examined, if a contradiction is reached for *any* variable i.e., the lower bound exceeds the upper bound, then the system is independent.

   For example, if
   $$\begin{aligned} 1 \leq & \quad t_1 & \leq 10 \\ 1 \leq & \quad t_2 & \leq 10 \\ 1 \leq & \; t_2 + 9 & \leq 10 \\ 1 \leq & \; t_1 - 10 & \leq 10 \end{aligned}$$

   are the bounds that result after application of the extended GCD test, the

bounds from the first and last constraint reduce to

$$11 \leq t_1 \leq 10, \text{ a contradiction.}$$

3. **Acyclic test.** If at least one constraint has more than one variable, the test mentioned above cannot be applied. The constraints involving more than one variable are represented by a directed graph. For each variable t_i, two nodes are created, labeled $+i$ and $-i$. An inequality constraint is rearranged and expressed as a bound for one of the variables. For instance, a constraint

$$a_i t_i - a_j t_j + \cdots \leq \cdots$$

is rewritten as

$$a_i t_i \leq \cdots + a_j t_j$$

In the graph, an edge is added from node $+i$ to node $+j$ if both a_i and a_j are greater than zero. This step is repeated for all the variables that occur in a constraint.

If the resulting graph has no cycle, a node with no incident edges is chosen and the value of its lower bound calculated (by some other method, say the Single Variable per Constraint test). The procedure is repeated for other nodes in the graph that have no incident edges or that have all their predecessor nodes set to some lower bound. The system is independent if the process of elimination results in a contradiction, otherwise it is dependent.

4. **Loop residue test.** For cyclic graphs, weighted edges are added, the weight being equal to the constant term in an inequality. If the sum of the weights of all edges in the cycle is a negative value, the system is independent. We remark that such a *sum* denotes the transitive closure and a negative value of the sum implies that $t_i \leq t_i - sum$, a contradiction.

By memoization, we mean that the results obtained on a previous invocation of some dependence test are stored so that when similar inputs are encountered later, we need not go through the entire process of dependence checking again. Memoization can lead to tremendous savings in time since typical programs have similar array access patterns throughout the program; in fact even loop bounds happen to be the same. Maydan *et al.* [10] have shown results which indicate that employing memoization can offset the extra expense incurred by applying these exact tests.

2.6.2 The Omega test

Pugh [14] takes an even stronger line that a perfectly accurate test can be employed for resolving data dependence. The Omega test [14] has worst-case exponential time complexity but Pugh claims that dependence analysis in real world programs seldom, if ever, encounter such a case.

Inequality constraints with the dependence equation to be solved can be viewed as a geometric problem and each iteration of the Omega test reduces the problem to an integer programming problem in fewer dimensions. A technique called *Fourier-Motzkin variable elimination* is used for this purpose, which finds the $(n-1)$ dimensional 'shadow' cast by projecting an n-dimensional object. Given the dependence equation in n variables, the constraints define an object (region) in n-dimensional space. A variable is eliminated by projecting the object along the axis representing that variable. Ideally, to maintain equivalence between the original problem and the refined problem, every integer point in the shadow should have an integer point in the object that was projected and vice-versa—such a shadow is termed an *integer shadow*. Since ensuring this is difficult, we differentiate two types of shadows,

a dark shadow that ensures that every integer point in it has an integer point

in the object above it and

a real shadow where integer points may have no corresponding integer point in the object above.

These shadows now represent the problem in $(n-1)$ variables and are interpreted as follows.

- No integer point in the real shadow, would mean, that there are no integer solutions to the integer set of constraints
- An integer point in the dark shadow implies the presence of integer points in the object that was projected.
- When the real and integer shadows do not coincide, if an integer solution exists, it must be closely nestled between an upper and lower bound of an inequality constraint. Hence a series of planes that are parallel to a lower bound and 'close' to it are considered (the number of such planes considered is a function of the coefficients of the eliminated variable in the different constraints). If an integer solution exists, it must lie on one of these planes. Refer [14] for more details.

The last of these steps can particularly be very time consuming, since the number of planes can at times be linearly dependent on the (absolute) value of the coefficient of the variable being eliminated. Usually, such cases are rare.

2.7 A Unified Approach to finding all dependences with minimal inaccuracies

In this section, we consider heuristics to minimize the inaccuracies caused due to inexact dependence checking. The actual test used for dependence analysis is

of no consequence here—these heuristics can be used with any test to improve it's efficiency.

Hierarchical dependence testing

For a direction vector of length n, there are as many as 3^n possible combinations. A linear search of such a large space would turn out to be expensive, particularly since a dependence exists in very few directions. Cytron and Burke [4] have suggested a scheme of hierarchically testing for all direction vectors which is patterned on the lines of a ternary search.

The idea is to represent the various direction vectors in the form of a tree with the root of this tree representing the most general direction vector, $(*, *, \ldots, *)$. At a level l of the tree, the direction vectors of the previous level are expanded at position l of the vector. The tree for a two-way nested loop is shown in the Figure 2.1. We can view each child in the tree as a refinement of its parent; thus if independence is shown for any vector, then the subtree rooted at that node need not be explored.

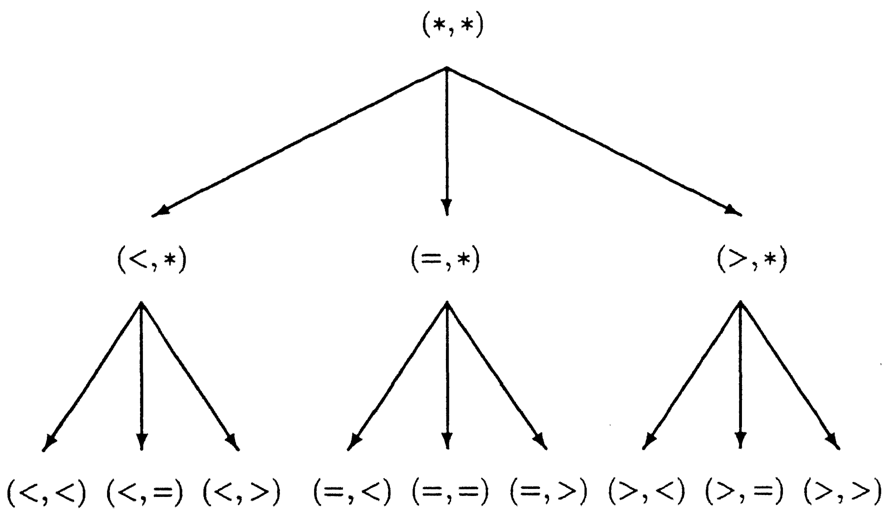


Figure 2.1: Hierarchy of direction vectors for a two-way nested loop

Finding both anti-dependences and flow-dependences together

Direction vectors indicate the direction in which the data flows from one iteration to another. Recall that the direction '>' implies that *data computed by some iteration* (i) *of a loop index variable is used by an* earlier *iteration* (j) *of that loop index variable.* Since the iteration j has already elapsed, this dependence does not hold.

Definition. A direction vector representing a dependence between two statements S_1 and S_2 is plausible if the instance of S_1 executes before the corresponding instance of S_2. Otherwise, it is implausible.

Example 2.2 For the two loop case, the implausible direction vectors are

$$(=,>),\ (>,=),\ (>,<),\ (>,>).$$

If S_2 depends on S_1 in $(=,=)$ and S_2 is statically before S_1, then $(=,=)$ is also an implausible one. ■

However, implausible direction vectors should not always be discarded as the following lemma shows.

Lemma 2.1 For a dependence from S_1 to S_2 under an implausible direction vector Z, a corresponding plausible direction vector Z' exists between S_2 and S_1 which is the complement of Z. Z' is calculated from Z by modifying each element of the direction vector as follows.

Element of Z	**Corresponding element of Z'**
$<$	$>$
$=$	$=$
$>$	$<$

Proof. Let $f(i_1, i_2, \ldots, i_n) = g(j_1, j_2, \ldots, j_n)$ be the diophantine equation. Then

$$f(i_1, i_2, \ldots, i_n) - g(j_1, j_2, \ldots, j_n) = 0 \qquad (A)$$

Let Z represent a direction vector under which a dependence exists from S_1 to S_2. We can rewrite (A) as

$$g(j_1, j_2, \ldots, j_n) - f(i_1, i_2, \ldots, i_n) = 0 \qquad (A')$$

If in (A), i_k is compared with j_k to produce a component of Z, then in (A'), j_k is compared with i_k to produce the corresponding complement. (A') is of the form of the dependence equation between S_1 and S_2 and hence the dependence between S_1 and S_2 exists with direction vector Z'. Thus, if Z is implausible, its complement Z' is plausible. ∎

Theorem 2.5 A flow-(anti-) dependence under a plausible direction vector is equivalent to an anti-(flow-) dependence under an implausible direction vector.

Proof. Refer [4] ∎

Reducing inaccuracies by filtering direction vectors

Recall that one of the inaccuracies that result from the testing procedure is introduced by subscript-by-subscript checking. A solution to the linear diophantine equation may exist in one subscript but not in another. If a dependence exists with a particular direction vector, it should hold in *every* subscript of the two references being tested. A number of false dependence vectors can thus be eliminated if they do not occur in any subscript.

A loop independent recurrence cannot exist, since it implies that a statement depends on itself in the same iteration—something impossible.

Example 2.3 As an example of how these different heuristics may be applied to rule out some spurious dependences, consider the matrix multiplication program segment shown below.

```
      DO I = 1, N
         DO J = 1, N
            DO K = 1, N
S:             C(I,J) = C(I,J) + A(I,J)*B(J,K)
            ENDDO
         ENDDO
      ENDDO
```

Applying Banerjee's and GCD tests for each subscript, we get the dependences shown.

Subscript 1	**Subscript 2**	
$(=,<,<)$		
$(=,<,=)$		
$(=,<,>)$		
$(=,>,<)$		
$(=,>,<)$		
$(=,>,=)$		
$(=,>,>)$		
$(=,=,<)$	$(=,=,<)$	
$(=,=,=)$	$(=,=,=)$	(*Short-listed direction vectors*)
$(=,=,>)$	$(=,=,>)$	
	$(<,=,<)$	
	$(<,=,=)$	
	$(<,=,>)$	
	$(>,=,<)$	
	$(>,=,=)$	
	$(>,=,>)$	

It is seen that most of the direction vectors for which a dependence has been detected are spurious ones. By intersecting the set of direction vectors obtained for each subscript, we can filter out many of the spurious dependences.

Further, the single statement loop independent recurrence $(=,=,=)$ cannot exist. Hence the only two direction vectors that remain are $(=,=,<)$ and the complement of the infeasible direction vector $(=,=,>)$, namely $(=,=,<)$, both representing a recurrence in S. ■

Chapter 3

Loop Restructuring Techniques

Data dependence analysis techniques check only the existence of a dependence between two array references. Extracting parallelism from a loop involves viewing the dependences from a different perspective—in the context of the loop nest as a whole[1]. A number of techniques have been developed to extract parallelism from a loop; their utility depending on the type of dependences that exist. As loops are the main source of parallelism, these restructuring techniques work on the *data dependence graph* (DDG) for a loop nest. The restructuring techniques mentioned here have been well described in [2], [12], [13] and [19].

We restrict ourselves to the analysis of `DO`/`FORALL` loops in Fortran D. The `FORALL` loop has the same syntax as that of a Fortran `DO` loop. All the loop iterations can run in parallel but the body of the loop is executed sequentially in each iteration.

[1]This process can be likened to performing optimizations like dead code elimination, induction variable elimination etc. after the global data flow analysis phase

3.1 Statement reordering

One of the basic tasks in generating parallel code is to detect if the various dependences of a loop nest form a cycle in the data dependence graph (DDG). Cyclic graphs cannot be trivially parallelized and the cycles need to be "broken" or "shrunk".

In contrast, acyclic graphs are much easier to handle. Intuitively, a dependence imposes an ordering on the execution of statements—execution of the statement corresponding to a node in the DDG will have to be curbed until all statements on which it depends finish execution. Thus, if we reorder the statements and execute them in the order imposed by the dependences, all dependences would be satisfied even if the iterations themselves were to be executed in parallel. The first statement in this new ordering is the one which is dependent on no other statement.

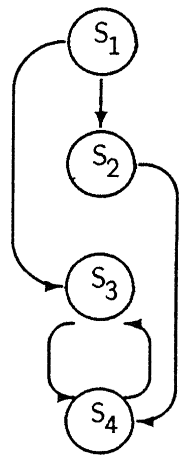
Notice that cyclic graphs consisting of a single component have no node with zero in-degree. Hence, statement reordering can be applied only to acyclic graphs.

3.2 Loop distribution (fission)

The presence of a cycle in a dependence graph does not imply the participation of all vertices of the DDG in the cycle. If vertices in a cycle do not inhibit parallelization of other statements, the loop can be fissioned in such a way that the original loop is distributed around each part.

Example 3.1 Consider the following program segment and its associated data dependence graph.

```
    DO I = 1, N
S1:    A(I) = E(I) + 1
S2:    B(I) = F(I) * A(I)
S3:    C(I+1) = A(I) + D(I)
S4:    D(I+1) = B(I) + C(I)
    ENDDO
```



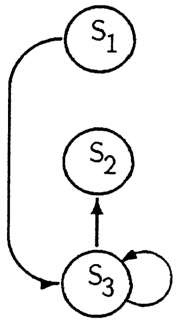
$\Downarrow$

```
DO I = 1, N                            FORALL I = 1, N
   A(I) = E(I) + 1                       A(I) = E(I) + 1
   B(I) = F(I) * A(I)                    B(I) = F(I) * A(I)
ENDDO                                  ENDFOR
DO I = 1, N                   =>       DO I = 1, N
   C(I+1) = A(I) + D(I)                  C(I+1) = A(I) + D(I)
   D(I+1) = B(I) + C(I)                  D(I+1) = B(I) + C(I)
ENDDO                                  ENDDO
```

Loop fission is effective only when the DDG for the loop nest in question has been split into more than one component, at least one of which is acyclic. Loop fission may often have to be accompanied by statement reordering in the loop which has an acyclic DDG. A condition that should necessarily be met by the loops resulting from fission is that no statement in the first loop resulting from fission should be dependent on any statement of the second loop.

For instance

```
    DO I = 1, N
S1: A(I) = B(I)**2
S2: D(I) = B(I)*C(I-1)
S3: C(I) = C(I-1) + A(I)
    ENDDO
```



cannot be transformed to

```
    DO I = 1, N
S1: A(I) = B(I)**2
S2: D(I) = B(I)*C(I-1)
    ENDDO
    DO I = 1, N
S3:    C(I) = C(I-1) + A(I)
    ENDDO
```

since the dependence $S_3 \rightarrow S_2$ is violated in the transformed loop.

Hence, we have the following lemma, due to [19],

Lemma 3.1 (Wolfe[19]) If a loop L contains statements S and T, where S statically precedes T in the loop and $T \rightarrow S$ is a data dependence, then the loop cannot be fissioned at any point between S and T.

A valid 'fission point' in the example above would be one between S_1 and S_2.

3.3 Breaking a cycle of dependences

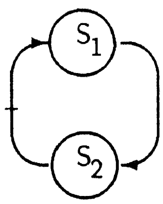
The methods described hitherto have considered grouping together statements that are not involved in any cyclic dependence and executing them in a FORALL loop. In most cases, a cycle of dependences inhibits concurrentization of the loop, unless suitable transformations are made to it.

3.3.1 Node splitting

Of particular interest are anti- and output-dependences which are part of a cycle. In a sense these are pseudodependences. An anti-dependence for instance, prevents a statement from updating a value of a variable merely because the value of this variable is to be accessed later. If the original value of the variable could be saved elsewhere and accessed when needed, such a dependence would cease to exist.

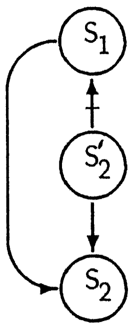
Example 3.2 In the example shown below, S_2 is anti-dependent on S_1 caused by the references A(I+1) in S_2 and A(I) in S_1.

```
     DO I = 1, N
S1 : A(I) = B(I) + C(I)
S2 : D(I) = A(I-1) * A(I+1)
     ENDDO
```



$\Downarrow$

```
     DO I = 1, N
S2':   TEMP(I) = A(I+1)
     ENDDO
     DO I = 1, N
S1 :   A(I) = B(I) + C(I)
S2 :   D(I) = A(I-1) * TEMP(I)
     ENDDO
```



■

Breaking an anti-dependence edge involves introducing a new statement. However, since the new statement itself is not dependent on any other statement, *it does not create a new cycle*; hence the amount of parallelism can never decrease.

Similarly, an output-dependence may be removed by promoting the array involved in the dependence to a higher dimension [19].

3.3.2 Breaking cycles of data dependences

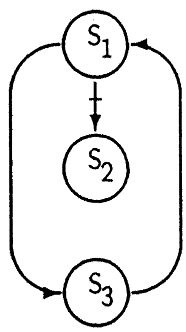
A cycle of data dependences is, in contrast to other false dependences much harder to break. Techniques for breaking such cycles are usually drawn from common optimization methods and it is difficult to determine a priori whether they will reap any benefits.

Statement substitution

Replacing the use of a variable by the r-value of the expression which is assigned to the variable when it is defined may delete an edge of a cycle as the example below illustrates.

Example 3.3 Consider the following program segment and its associated DDG.

```
     DO I = 1, N
S1:     A(I) = C(I) + B(I)
S2:     C(I) = E(I)
S3:     B(I+1) = A(I) + 2
     ENDDO
```



⇓ *Statement reordering*

```
     DO I = 1, N
S1:     A(I) = C(I) + B(I)
S3:     B(I+1) = A(I) + 2
S2:     C(I) = E(I)
     ENDDO
```

```
     DO I = 1, N
S1:     A(I) = C(I) + B(I)
S2:     B(I+1) = (C(I)+B(I)) + 2
S3:     C(I) = E(I)
     ENDDO
```

The cycle between S_1 and S_3 has been broken after statement substitution. Observe that substitution without reordering the statements would lead to a violation of the semantics of the program in this example. ■

Statement splitting

Analysis of subexpressions may help in tapping parallelism of a finer grain. This makes sense as a dependence in a complicated expression may be due to a single array reference and it would be rather unfair to condemn an entire statement if this dependence is part of a cycle. Statement splitting extracts expressions that do not cause any dependence and puts them in a parallel loop. While cycles are not broken, all references in a statement not involved in any dependence are factored out; thus it can be viewed as the complement of node splitting. Refer [12] or [19] for more details.

3.4 Loop interchanging

Generating parallel code after analyzing a nest for loop dependences is not determined just on the basis of a cycle in a graph. By definition, if the depth of the dependence is not equal to the depth of the loop nest, then disregarding some of the outer loops by holding them constant would remove the dependence. The number of loops that are to be held constant here are d, where d is the depth of a dependence. If d happens to be the maximum depth among *all dependences* in the loop nest, then $l - d$ gives the number of loops that may be executed in parallel. An interesting question arises now whether the amount of parallelism reflected in $l - d$ can be increased by decreasing d.

Example 3.4 The code given below performs matrix multiplication

```
DO I = 1, N
   DO J = 1,N
      DO K = 1, N
         C(I, J) = C(I, J) + A(I, K) * B(K, J)
      ENDDO
   ENDDO
ENDDO
```

The depth of the dependence in the original code was 3 and the amount of parallelism exposed nil. By interchanging loops `J` and `K` initially followed by loops `I` and `K`, we can decrease the depth of the dependence to 1 and increase the number of parallelizable loops to 2.

```
DO K = 1, N
   DO I = 1,N
      DO J = 1, N
         C(I, J) = C(I, J) + A(I, K) * B(K, J)
      ENDDO
   ENDDO
ENDDO
```

There is no dependence now at a depth > 1. Consequently, the loops `I` and `J` can be parallelized. ■

Loop interchanging alters the order in which elements of an array are computed. Here, semantics can be preserved only if the values necessary for computation of a statement have been evaluated prior to execution of the statement, in the interchanged loopnest. In particular, interchanging two loops which contribute directions "<" and ">" to the same direction vector cannot be interchanged, the intuitive reason being that in the interchanged loop nest, a dependence is imposed on a variable whose value is to be computed in a later iteration. We elaborate on this in more detail in the feasibility tests discussed in the ensuing chapter.

Enhancing the parallelism by cycle shrinking

While loop interchanging enables parallelization of all loops that are nested at a depth $> d$, the maximum depth of any dependence, it still leaves certain iteration instances in the loop at depth d in the nest which are free of any dependences among themselves. Such iterations could be executed concurrently.

Example 3.5 Consider the loop

```
DO I = 3, N
   DO J = 5, N
      A(I, J) = B(I-3, J-5)
      B(I, J) = A(I-2, J-4)
   ENDDO
ENDDO
```

In loop I, the distance of the dependence between the two references to A is 2 and between the two references to B is 3. Any two iterations I and I+1 are free from dependences among themselves (iteration I+2 is however dependent on iteration I); they could be iterated in parallel.

```
DO I = 3, N
   FORALL I1 = I, I+1
      FORALL J = 5, N
         A(I, J) = B(I-3, J-5)
         B(I, J) = A(I-2, J-4)
      ENDFOR
   ENDFOR
ENDDO
```

■

If p is the minimum of all distances in component d of different direction vectors, then the loop at depth d in the nest can be shrunk by a factor of p. This approach to shrink cycles is termed as selective shrinking. Besides this, [13] also describes other approaches to cycle shrinking.

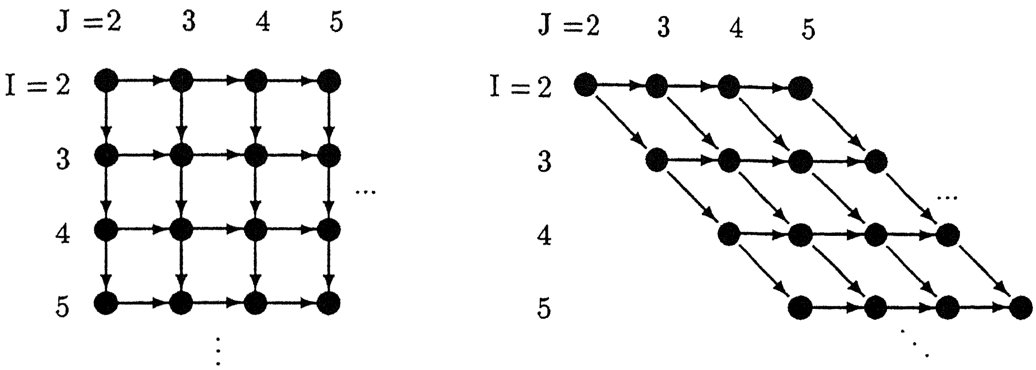


Figure 3.1: ISDG for the Jacobi problem before and after skewing

3.5 Loop skewing

When a dependence exists at every level in the loop nest, interchanging loops has no effect since it cannot increase the depth of the dependence. Altering the shape of the *iteration space dependence graph* (ISDG), which is a depiction of the dependences across different iterations, could remove some of the inter-iteration dependences, enabling the loop to be partially parallelized.

The entire ISDG appears to have been sheared/skewed by a certain amount about the axis line `I=1`. The loop index variable J's loop bounds have been modified in the new scenario, but as each iteration of `J` is free from any dependence, they can be iterated in parallel. To skew a loop index `J` with respect to another index `I` by a factor `f`, we

- add the expression `I*f` to the lower and upper bounds of the loop index variable `J`
- replace every reference to `J` in the loop body by `(J-I*f)`

Example 3.6 The Jacobi problem's code is shown below.

```
DO I = 2, N-1
   DO J = 2, N-1
      A(I, J) = (A(I+1, J) + A(I-1, J) + A(I, J+1) +
                 A(I, J-1))/4
   ENDDO
ENDDO
```

Skewing J wrt I by a factor 1, we get

```
DO I = 2, N-1
   DO J = I+2, I+N-1
      A(I, J-I) = (A(I+1, J-I) + A(I-1, J-I) + A(I, J-I+1)
                  + A(I, J-I-1))/4
   ENDDO
ENDDO
```

The ISDGs for the original and transformed programs are shown in Figure 3.1. With no dependence now at a depth 2, we may interchange the loops enabling parallelization. ■

By carving out a rhomboidal region from the original square region, loop skewing removes the dependence at a certain level. Finding the appropriate factor by which the loop should be skewed can be viewed as a problem of finding that hypothetical wavefront/hyperplane which is dependence-free. The dashed lines in Figure 3.1 above represent the *waves* in each parallel execution. Refer [21] for more details.

3.6 Other issues in restructuring

3.6.1 Decreasing overheads of a FORALL by loop jamming

One problem with FORALL loops is the overhead involved in setting them up. There is little justification in parallelizing a loop whose body is too small to

warrant such an "optimization"; in an extreme case when the loop size is very small ($\simeq$ 3-5), overhead becomes so high that the serial loop itself would have performed better. Merging bodies of two adjacent loops with identical bounds reduces the overheads considerably since the setup time for the merged loop is reduced by half. To ensure semantic correctness, there should be no cross-iteration dependence at any level of the loop nests between references in the two loop bodies.

```
FORALL I = 1, N                              FORALL I = 1, M
   β1(I)                                        β1(I)
ENDFOR                     ⇒                    β2(I)
FORALL I = 1, N         Loop jam             ENDFOR
   β2(I)
ENDFOR
```

3.6.2 Handling WHILE loops

The iterator construct WHILE differs from the DO loop in that the exit from a WHILE loop could depend on an arbitrary relational expression. The exact number of iterations for which a WHILE loop would iterate cannot be known in most cases at compile-time.

This suggests that if a WHILE loop has an induction variable, and the relational expression controlling entry into the body of the loop is a simple function of this variable, it may be possible to convert such a WHILE loop to a DO loop, on which the restructuring techniques described earlier can be applied. Wu and Lewis [22] take this approach to parallelize WHILE loops. Loops that are built from unstructured constructs like GOTO and conditional jumps are extremely difficult to parallelize since they involve an in-depth analysis of the control flow graph[2].

[2] A control flow graph depicts the flow of control between different statements in the program. The control flow graph has been dealt with in section 5.6.

Chapter 4

Generating parallel code

In the last chapter, we saw the various techniques by which it was possible to restructure a serial loop into a parallel one. We now consider how these techniques are coordinated and applied in a manner so as to get the best out of the restructurer. The order in which these transformations are applied greatly affects the parallelism obtained and this has been a topic of much theoretical interest of late [18]. The algorithm given here is however based purely on heuristics and the merits of each transformation. We study the feasibility tests that are necessary for the application of a transformation, an essential step to not only check the utility of applying a transformation but also limit the generation of redundant code. We clarify that, by *generating parallel code*, we mean the parallel equivalent of the input program in a high-level language. Finally, the importance of user directives and data distribution in controlling the code generated by the restructurer is highlighted.

4.1 The algorithm

Inherent in the discussion of restructuring techniques presented in the last chapter is the concept of a π-block. A π-block represents a strongly connected com-

ponent of the DDG (a cycle or a single statement). In principle, the algorithm works on the same lines as that of the vector code generation algorithm mentioned in [2], with the added capability to break dependence cycles wherever possible and coalesce parallel loops generated for adjacent π-blocks whenever legal. Each loop nest is considered at a time and code subsequently generated for each π-block. Thus, a π-block is the basic unit for parallel code generation.

Algorithm 4.1 Parallel_codegen.

```
/* find strongly connected components of the DDG and store in π-block */
find_scc(ddg, π_block);
topological_sort(pi_block);
for (i=0; i<components; i++) {
    if (cycle_detect(π_block[i])) {
       if (node_split(π_block[i])) {
          topologically sort the subblocks generated by converting
          π_block[i] to an acyclic one
          for all subblock[j], a subblock of π_block[i]
              loop_distribute(subblock[j], PAR)
       }
       else {
          loop_distribute(pi_block[i], NO_PAR)
          loopinterchange(pi_block[i])
       }
    }
    else loop_distribute(pi_block[i], PAR)
    mark π_block[i]
}
delete vertices in all marked π_blocks from the DDG
```

To ensure that dependences are satisfied when statement(s) in the π-block is (are) executed, we need to generate code in the order imposed by the dependences. A topological sort of π-blocks is done to ensure that statements are reordered correctly. A dependence edge between two π-blocks π_1 and π_2 imposes a "<" relationship between them and is used to determine which of them is the predecessor of the other during a topological sort. Preference is given to

distribute the loops around π-blocks when they are detected to be acyclic—the ideal case. For π-blocks that represent a strongly connected component, attempts are made to break cycles or interchange loops to increase the degree of parallelism. At each stage, we attempt to coalesce (fuse) the loops generated with that for a previously generated π-block whenever legal.

A few comments would be in order now. `cycle_detect()` detects the presence of a cycle in π-block. Loop distribution converts the distributed loop to a parallel one when requested to do so with the PAR parameter. Secondly, the algorithm attempts to coalesce the distributed loop with its predecessor in the abstract syntax tree when legal. We observe that although statement reordering has not been *explicitly* done, it is automatically handled by topological sort, loop distribution and loop coalescing. Application of each transform is preceded by a feasibility test which deliberates on the wisdom of applying that restructuring technique. When node splitting is successfully applied to break the cycle in a π-block, a number of subblocks may be generated. The loopnest needs to be distributed around each acyclic subblock. The loop fusion routine is called by the loop distribution routine itself.

Other restructuring techniques are easy to incorporate in the algorithm. For instance, cycle shrinking is best done after application of loop interchange while loop skewing would be more productive before loop interchange (refer the discussion in Chapter 3).

4.2 Feasibility tests for restructuring techniques

A prerequisite for any restructuring technique is to prevent or at least minimize the generation of redundant code. And of course, check if all necessary conditions for the transformation to be applied are satisfied. For instance, distributing

a parallel loop around a π-block can be done only if the block is acyclic. Transformations generating new statements (such as node splitting) need to make sure that they will finally yield some parallelism worth the effort. The employment of fairly robust feasibility techniques and their well-defined interfaces with the corresponding transforms is another feature of FRAMES.

4.2.1 Loop distribution

As loop distribution is used for the twin purposes of generating code for an acyclic π-block and for a π-block containing a cycle, the `cycle_detect()` procedure in the algorithm would itself suffice. Loop distribution is always applied with the objective of removing obstacles for generating parallel code for the remaining π-blocks. Further, since it is immediately followed by loop coalescing, there can never be any redundant `DO/FORALL` loops generated.

4.2.2 Loop fusion

Adjacent loop bodies can be fused/jammed if no two statements in the two bodies are involved in a dependence that modifies the value of a variable used by the other. The algorithm fuses a loop only with a similar one (i.e., a loop with the same index variable and identical bounds) that had previously been generated in the same invocation of `Parallel_codegen`. As dependence analysis and the subsequent topological sort ensures these conditions, it suffices to check if the loops have the same index variable, loop limits and if they are not involved in any cross-iteration forward dependence.

Loop fusion is an optimization aimed at increasing the length of the loop body; this rules out the possibility of redundant code being generated.

4.2.3 Node splitting

As mentioned in the previous chapter, node splitting is done to remove anti-dependences. A simple and sufficient feasibility test to ensure that redundant node splittings are eliminated/minimized is considered below. For an anti-dependence edge between two statements S_1 and S_2 that is part of a cycle, it would be worthwhile to break the edge only if there is no other path consisting purely of data dependence edges/scalar dependences from S_1 to S_2 —edges, which are in practice, seldom possible to break. This amounts to tentatively deleting an anti-dependence edge from the graph and verifying if the two vertices in question have become part of different components. Since deletion of an edge may render other deletions to become redundant, we need to take care of this. Algorithm `feas_split` does *not* detect the minimum number of edges to be deleted during node splitting; rather, it ensures that in the sequence of edges that are marked for deletion, none are redundant.

Assume that all parallel edges between two vertices in the graph have been replaced by a single edge between the two. We can classify the anti-dependence edges that are processed by `Feas_split` to be in one of the following states—marked for node splitting, unmarked or simply discarded. Discarded edges are those edges which need not be considered during node splitting since they are 'covered' by other anti-dependence edges. The sets S_u and S_m are used in the algorithm to summarize such marker information. An edge in S_m is marked, that in S_u is unmarked while an edge in neither is considered to be discarded. To begin with, all edges are unmarked and when the algorithm ends, S_m gives the set of edges whose removal by node splitting would lead to a productive gain in parallelism. During node splitting, the source node of each such edge is a candidate for node splitting.

Algorithm 4.2 Feas_split

/* Initialize the set of all unmarked anti-dependences and the set of all
   marked anti-dependences */
S_u = {all anti-dependence edges}
$S_m = \emptyset$
repeat
  Let e be an anti-dependence edge in S_u between say, S_i and S_j
  Check if this anti-dependence edge is still part of a cycle consisting of no
    marked edges (edges in S_m)
    and no discarded edges (edges in neither S_m nor S_u)
  if so, trace a path consisting of no anti-dependences (marked or unmarked)
    between S_i and S_j
  /* Mark this edge if needed */
  if there exists no such path,
    $S_m = S_m + \{e\}$
  $S_u = S_u - \{e\}$ /* delete processed edge from S_u */
until ($S_u = \emptyset$)

4.2.4 Loop interchanging

Loop interchanging too never generates redundant code as only the order of the nest is changed. Given an n-way perfectly nested loop, the optimal ordering of the n loops requires generating each combination and testing for the best one. The 'best' ordering is usually the one that causes the depth of any dependence to be minimum. The feasibility test for this involves pair-wise permutations; so the current permutation's validity can be tested by comparing the directions induced on each dependence by these two loops. The possible feasible direction vectors before and after interchanging (neglecting all other loops temporarily) are :

Initial direction vector	After interchanging	Comments
$(<,=)$	$(=,<)$	Valid
$(<,>)$	$(>,<)$	Invalid
$(=,=)$	$(=,=)$	Valid
$(=,<)$	$(<,=)$	Valid

The second one is deemed to be invalid for reasons mentioned in the previous chapter. In the ISDG, these direction vectors appear as shown in Figure 4.1. Notice that in the interchanged loop, the direction vector $(>,<)$ would imply a dependence on a value that is to be computed in a later iteration; hence, the interchange is illegal. For a formal proof, refer [2] or [19].

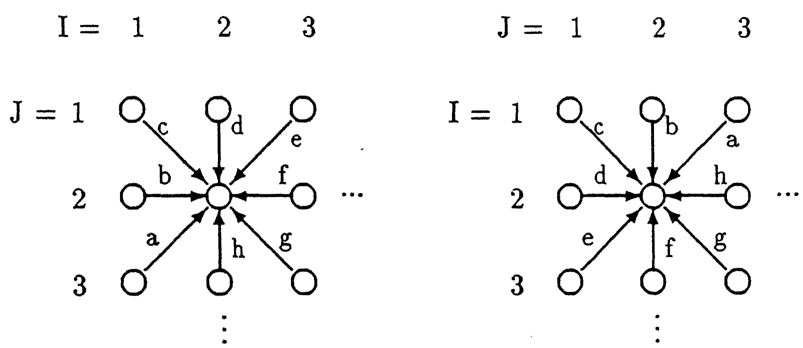


Figure 4.1: Dependence patterns in a loop—dependences e, f and g are destroyed when the loops are interchanged and J becomes the outer loop

The algorithm **Exchange** given below is from [19].

Given a particular loop ordering (`loopnest`) and the position of a loop (`r`) in the nest, the algorithm finds all possible combinations of `loopnest` without interchanging the loops at positions `r` and `r+1` and after interchanging the loops at positions `r` and `r+1`. The primary purpose of `adjloops` is to ensure that loops that have already been adjacent once, are not brought adjacent to each other again in some other permutation. The current loop ordering is better than

Algorithm 4.3

```
function Exchange(loopnest, bestlooporder, r, adjloops)
/* loopnest      : given loop nest
 * bestlooporder: the optimal nest outputted by Exchange
 * r             : current loop that we are looking at in loopnest
 * adjloops      : set of pairs of loops that have already been adjacent
 */
{
    if (r == depth(loopnest)) {
      if (depth_of_dep(loopnest) ≥ depth_of_dep(bestlooporder)) {
        bestlooporder ← loopnest
        return(1);
      }
      else return(0);
    }
    adjloops = adjloops ∪ (loopnest[r], loopnest[r+1])
    /* all possible combinations without interchanging loopnest[r]
     * & loopnest[r+1] */
    Exchange(loopnest, bestlooporder, r+1, adjloops, list)
    adjloops = adjloops − {(loopnest[r], loopnest[r+1])}
    if (is_interchangeable(loopnest, r, adjloops)) {
      /* interchange the values in loopnest[r] and loopnest[r+1] */
      newloopnest ← loopnest
      newloopnest[r] ↔ newloopnest[r+1]
    }
    else return
    if (r == 1) {
      if (Exchange(newloopnest, bestlooporder, 2, adjloops, list) == 0)
        return
    }
    adjloops = adjloops ∪ (newloopnest[r], newloopnest[r+1])
    if (Exchange(newloopnest, bestlooporder, r-1, adjloops, list) == 0)
    adjloops = adjloops − {(newloopnest[r], newloopnest[r+1])}
}
```

the best seen so far if the maximum depth of any dependence in this ordering is less than that of the latter. Function `is_interchangeable()` verifies if the loop interchange is legal and if the loops are in `adjloops`.

4.3 Techniques to improve detection of parallelism

Throughout this report, we have stressed the point that inaccuracies in the restructurer, though on the conservative side, are inevitable. It is possible to neglect some dependences in an attempt to gain more parallelism, if their effect is known. Obviously, such decisions lie in the domain of the user and some mechanism needs to be provided that would enable the user to interact with the compiler.

4.3.1 Directives

One way to force parallelization of a loop is to give a compiler directive to this effect. The directives supported in the prototype of FRAMES are modeled on those supported by the optimizing Fortran compiler for CONVEX *C-220*([23]. Every directive statement has the following format :

```
C$DIR <directive> [optional parameters for the directive]
```

The C$DIR should begin from the first column in a line to avoid confusion with other Fortran77/Fortran D statements. The following directives are currently supported by FRAMES.

`FORCE_PARALLEL` forces parallelization of the loop immediately following this statement. This is equivalent to replacing the `DO` loop of interest with a `FORALL` loop.

NO_PARALLEL forces serial execution of the loop immediately following this statement. AS FRAMES never incorrectly parallelizes a loop, the necessity of including this directive may appear to be perplexing. However, for very small loops, it may be more beneficial to execute it serially if the overheads of parallel execution of the loop are high. Such issues assume significance if the loop is part of a much bigger nest.

MAX_TRIPS *count* indicates the maximum number of iterations that the loop following this statement executes. If the upper bound for the loop index variable is unknown, the conservative assumption of infinity may introduce more errors in dependence analysis. The MAX_TRIPS directive is a step towards decreasing such errors.

Other useful directives that could be supported are

Task identification directive

The syntax of this directive is given below.

```
C$DIR BEGIN_TASK
   statements
C$DIR NEXT_TASK
   statements
C$DIR NEXT_TASK
   statements
      ⋮
C$DIR END_TASK
```

A task is a set of statements that are considered to be a unit of execution on one processor. Tasks can be run in parallel if there are no dependences between them. Usually, statements which are not part of any loop (the

'inherently sequential' part of a program?) are merged into one task. More on tasks can be found in [6] and [13].

`NO_SIDE_EFFECTS (procedure_1, ..., procedure_n)`

Function/procedure calls in a loop body can curtail parallelism detection if they contain any side effects. As Fortran supports only call by reference, changes to any parameter can have an effect on the rest of the loop body. `NO_SIDE_EFFECTS` tells the compiler to disregard any side effects even if they are present. The parameters `procedure_1` ... `procedure_n` are the procedures that are claimed to be free from any side effects. In effect, this directive minimizes the errors of interprocedural analysis ([4], [16]).

It should be noted that the use of some of these directives (such as `FORCE_PARALLEL` or `NO_SIDE_EFFECTS`) *may* alter the semantics of the program. It is the prerogative of the user, who has complete knowledge of the application at hand, to ensure that any such changes are inconsequential.

4.3.2 A back end to the user

While directives enable the user to have some control, even if rudimentary, on the parallelization of a program, enough information needs to be output by the compiler if it fails to parallelize a piece of code. Currently, FRAMES merely dumps those parts of the DDG that prevented parallelization of a loop body on to a file. The restructured program itself is printed out, enabling the user to view for himself the parallelized segments of his program.

4.4 Improving performance with data distribution

We digress at this point temporarily to consider a few aspects that are more germane to the architecture of a multiprocessor than restructuring. Many researchers ([5], [7]) contend that the real bottleneck in multiprocessing is the memory access speed and a clever distribution of array elements among the

processor memories is needed to circumvent this. To get a flavor of the problem on hand, consider an array of size 100 whose elements are to be initialized to zero. If ten processors work on this array, maximum efficiency is obtained when the elements accessed by the processors are in their local memories. With elements in a shared (global) memory or other processors' local memories, delays are introduced, either due to serialization of concurrent requests from multiple processors or due to wait times arising from message passing primitives in a distributed memory system. Recognizing the right distribution for an array is however something a compiler is not very adept at doing automatically [7]. This is partly because the nature in which arrays are accessed differ in various segments of a program and an efficient distribution that takes all these into account needs to be found. A unique feature of Fortran D is the rich support given to data distribution. [6] and [7] address the topic of data distribution with reference to scheduling iterations on multiple processors in much greater detail.

Chapter 5

Implementation Esoterics

Implementation of a concept often faces many an unforeseen obstacle[1]. We consider now the minutiae of implementation, details such as the intermediate representation (IR) used, the structure of each node in the IR, data structures adopted and such of those issues which would appeal to an implementor.

5.1 Intermediate representation used

The input program from a user needs to be parsed and represented in a form that is easy to manipulate during restructuring. An abstract syntax tree structure is ideal for such an application, in view of the large number of changes that occur to the tree's contents in the course of execution of the compiler.

Each node in the abstract syntax tree has the format shown in Figure 5.1.

Every node represents a syntactic unit which is identified by **node_type**. Typically, a node has other data that is related to it. For example, associated with a node representing a `DO` loop would be its loop index variable, the lower and upper bounds of the loop, the increment and a pointer to the body of the loop. The union `elements` stores such data or pointers to data that are closely as-

[1] "Between the idea and the reality lies the shadow", *T. S. Eliot*

```
struct node {
  int node_type;  /* discriminator for type of node */
  union {
    int   i_val;  /* integer value to be stored */
    float f_val;  /* floating point value to be stored */
    char  *s;     /* pointer to a character string */
    struct node *node_ptr ; /* pointer to another node */
  } elements[5];
  struct node    *next;  /* pointer to next node in tree */
  struct id_data *table_entry_ptr;
  int            misc;   /* any other misc data */
  int            label;  /* label of the statement, if any */
  int            statno; /* a unique statement number for each
                          * statement */
  /* control flow graph related fields */
  struct node   *t_edge; /* true edge */
  struct node   *f_edge; /* false edge */
  struct node   *u_edge; /* unconditional edge */
  /* data flow analysis related fields */
  int            df_entry;
  int            visited;
};
```

Figure 5.1: Structure of each node in the syntax tree

sociated with a node. Most other fields are self explanatory. The control flow graph is discussed in section 5.6. Data flow analysis, an indispensable part of any compiler, is addressed in [16]. The entries of relevant fields for some nodes is given below. For a complete description on the value of every field in each node, refer [15].

DO loop :

`node_type` = `IS_DO_NODE`

`elements[0].node_ptr` = pointer to a node representing the loop index variable

`elements[1].node_ptr` = pointer to a node representing the expression for the lower bound of the loop index variable

`elements[2].node_ptr` = pointer to a node representing the expression for the upper bound of the loop index variable

`elements[3].node_ptr` = pointer to a node representing the expression for the increment of the loop index variable between iterations

`elements[4].node_ptr` = pointer to a node representing the loop body

Function/procedure definition :

`node_type` = `IS_FUNCTION/IS_SUBROUTINE`

`elements[0].node_ptr` = pointer to a node representing the name of the function/procedure

`elements[1].node_ptr` = pointer to the parameter list (described below) for the function/procedure

Function call :

`node_type` = `IS_FN_CALL`

`elements[0].node_ptr` = pointer to a node representing the name of the function

`elements[1].node_ptr` = pointer to the parameter list for the function/ procedure

Parameter list for a function/procedure :

`node_type` = `IS_PARM_LIST`

`elements[0].node_ptr` = pointer to a node representing a parameter (expression or identifier)

`elements[1].node_ptr` = pointer to the next parameter in the parameter list

Assignment statement :

`node_type` = `IS_ASSIGN`

`elements[0].node_ptr` = pointer to a node representing the left hand side of the assignment

`elements[1].node_ptr` = pointer to a node representing the right hand side of the assignment

Integer constants :

`node_type` = `IS_INT`

`elements[0].i_val` = integer value to be stored

Identifier :

`node_type` = `IS_CHAR`

`table_entry_ptr` = pointer to the symbol table entry for the identifier

5.2 The parser

We have built a parser for Fortran D to test the restructurer. While it does not handle issues such as error recovery or features of Fortran D that are not germane to the task of restructuring (eg. complex data types), it performs the objective of generating an intermediate representation for the input program seen.

However, features like global variables (`COMMON`) and aliasing (`EQUIVALENCE`) are supported to realize the full power of interprocedural analysis. The grammar for the language accepted by our parser is given in Appendix B.

5.3 Analysis of array references

We have limited ourselves to the use of inexact tests in the dependence analysis of references to the same/aliased arrays. The tests that have been implemented are the GCD and Banerjee's tests and the heuristics of section 2.7.

The restructurer works by examining one loop nest at a time. A unique loop number is given to each `DO/FORALL` loop in the nest during a preliminary scan. Array references which refer to the same/aliased arrays are candidates for data dependence analysis and a list of all reference pairs that need to be tested for is formed.

Every pair of references is passed to the dependence testing module. This module identifies the type of region that is described by the loop bounds i.e.. trapezoidal/rectangular and appropriately applies the corresponding algorithm. A dependence is assumed with the direction vector $(*, *, \ldots, *)$ if either of the loop bounds are non-linear. The (cheaper) GCD test is applied before Banerjee's test.

The testing process proceeds by generating a direction vector and then checking if the two references are dependent in this direction. Subscript-by-subscript dependence checking is employed. The direction vectors for which the references are independent in any subscript are not used during dependence testing of subsequent subscripts. This ensures that the set of extracted dependence vectors is the intersection of the different direction vectors for each subscript. Finally the appropriate edges are added to the data dependence graph.

5.4 Data dependence graph

The data dependence graph (DDG) is used to represent the dependences that are detected between statements. An adjacency list form has been used to represent the DDG, in view of its capability to handle idiosyncrasies such as multiple dependences between two statements, no dependence between two statements etc. Every vertex in the graph has the following structure :

```
struct graph_struct {
  int vertex1;            /* statement number of statement */
  int loops[MAX_LOOPS];   /* loops enclosing a statement */
  struct node *statptr;   /* pointer to the statement that
                           * vertex1 represents */
  struct adjvertex *adjv;/* list of vertices that are adjacent
                           * to this vertex */
};
```

Every vertex in the graph, `vertex1` is assigned the statement number of the statement that it represents. Although the inclusion of `vertex1` may appear to be superfluous, particularly since a pointer to the statement (`statptr`) is also being stored, the vast number of situations it is needed justify this negligible tradeoff in space. (It is much more elegant to use just `vertex1` than to access `statptr->statno` every time it is needed) Loops surrounding a statement need to be stored as they are required in imperfectly nested loop nests. This enables one to calculate the loops common to two statements with ease. While `loops[0]` gives the number of loops enclosing a statement, `loops[1]` `...loops[loops[0]]` give the actual loop numbers that enclose it (recall that a unique loop number is assigned to each loop in the nest).

Thus, our DDG has the structure

```
struct graph_struct graph[MAX_VERTICES];
```

We note the following in the representation of an edge between two vertices. A dependence is caused by *two references* and transformations like node splitting explicitly examine the references when they splice a reference from an expression. Further, the direction vector itself needs to be stored since it is useful in transformations such as loop interchange. Compression of direction vectors[2] introduces errors (*spurious* dependences), something which we can ill afford, particularly since exact tests are not currently being conducted at this stage in the development of the compiler. Thus, despite the overheads involved, we are forced to keep direction vectors uncompressed. The type of the dependence (deptype) is used by the feasibility test for node splitting (see section 4.2.3).

```
struct adjvertex {
  int                vertex2;   /* vertex involved in the
                                 * dependence */
  int                ncomloops; /* loops common between this
                                 * statement & the one with
                                 * which it is involved in
                                 * a dependence */
  struct node       *ref1;      /* reference 1 involved in
                                 * the dependence */
  struct node       *ref2;      /* reference 2 involved in
                                 * the dependence */
  enum direnums      dvect[MAX_LOOPS];
                                /* direction vector of the
                                 * dependence */
  enum dependence    deptype;   /* type of dependence */
  enum marker        tag;       /* used during graph search */
  struct adjvertex *nextadjv;   /* the next adjacent vertex
                                 * in the adjacency list for
                                 * the vertex which caused this
                                 * dependence */
};
```

[2]By compression, we mean that if two vectors differ in only one component of the vector, then they can be merged. For instance, $(=,=,<)$ and $(=,=,=)$ can be compressed and represented as $(=,=,\leq)$. Refer [4] for more details.

where the enumerated data types used are

```
enum direnums   {LESS, EQUAL, GREATER, STAR}
enum dependence {IS_SCALAR, IS_WEAK, IS_FLOW, IS_ANTI,
                 IS_OUTPUT, IS_CONTROL}
enum marker     {OLD, NEW}
```

We stress that `vertex2` is the statement number of a vertex involved in the dependence, and *not* the actual position of this vertex in `graph`. Field `ncomloops` is redundant since it can be obtained by comparing the `loops` of the two vertices involved in the dependence; but the frequency at which this value is needed justifies the inclusion of a separate field.

Computing the loops common to the dependent statements: Let `ptr` be a pointer to some `adjvertex` in the adjacency list of a vertex `v1`. Let `v1` occur at position `p` in `graph`. Then

```
graph[p].loops[1] ...graph[p].loops[ptr->ncomloops]
```

gives the common loops in the dependence.

Very efficient algorithms exist in the literature for traversal of a graph and other related applications like cycle detection. The complexity of these algorithms is linear in the number of edges of the graph ([1], [17]).

The enumerated data types listed above explain themselves. `direnums` are the possible directions that any direction vector can take. Similarly, `deptype` is used to indicate the dependence information. The necessity of `IS_SCALAR` and `IS_WEAK` dependence edges is justified in the following section.

5.5 Scalar dependence edges

Throughout this thesis so far, we have deliberately avoided using any scalar variables in the examples presented to prevent any obfuscation of the ideas

Apparently, parallelization is possible only if the scalar X is promoted to an array. FRAMES currently does not do this and takes the more conservative approach of serializing all statements in the loop that are involved in such cross-iteration scalar dependences.

Yet another nuance that needs to be taken care of is the presence of cycles in the DDG. Statements that form a dependence cycle may be involved in scalar dependences too. Going by the algorithm `Parallel_codegen` given in Chapter 4. the statements not involved in any scalar dependence could well be put into a parallel loop while those forming a cycle grouped into a serial loop—something we should not permit. To achieve this, we add edges from statements in the cycle to all its predecessors of scalar dependence edges. These edges are called *weak* edges, since they should not stand in the way of exploiting parallelism. Their inclusion is only to ensure the semantic correctness of the restructured loop. For instance, if the cycle could be broken by node splitting, then these weak dependences no longer hold and they should be deleted from the graph.

5.6 The control flow graph (CFG)

While the DDG depicts data relationships among statements, the flow of control among statements is equally important. This information is encapsulated in the control flow graph. In the presence of a conditional statement, the execution of statements in the different branches would be dependent on the value of the relational expression. There arise then, three possible edges in the control flow graph (refer Figure 5.1) —

- unconditional edge (u_edge), in normal sequential flow of control
- true edge (t_edge), the branch taken if the relational expression in the conditional evaluates to true

- false edge (f_edge), the branch taken if the relational expression in the conditional evaluates to false

For instance, in a syntax tree node with node_type IS_DO_NODE,

t_edge points to the first statement in the loop body
f_edge points to the statement immediately following the loop and
u_edge is unused

Such a superimposing of the CFG on the syntactic structure of the program is preferred as it enables one to access all fields immediately when required.

5.7 The restructuring techniques used

The transformations used in this phase ultimately determine the quality of the output generated by FRAMES. The algorithm Parallel_code_gen given in Chapter 4 has been used to coordinate the application of different transformations used namely, loop distribution, loop jamming, loop interchanging and node splitting for anti-dependences. The feasibility tests given in Chapter 4 determine the utility of applying a particular restructuring technique.

Most of the limitations in the transformations stage can be traced to the inadequacies of the analysis phase. The algorithms for loop skewing and cycle shrinking for instance, could not be implemented since the analysis phase currently does not have the capability to detect the distance of a dependence.

5.7.1 Handling conditional statements and jumps within loops

The last aspect that shall be touched upon is the presence of conditional statements in loops. A *control dependence* exists between a conditional statement and all statements that are under the control of one of its branches. [2] have

```
      DO 100 I=1, N
S1 :     IF (A(I) .LE. A(I+1))
S2 :        A(I+1) = A(I)
S3 :     B(I) = A(I+1)**2
100   CONTINUE
```

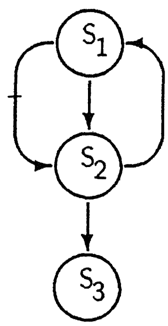


Figure 5.2: Depiction of control dependence edges

proposed a method called *if-conversion* which converts all control dependences to data dependences. This method is well suited for machines that have an underlying vector architecture since vector dialects like F8x or Fortran90 have *conditional assignments*, whose controlling conditions are explicitly stated in the statements themselves. As Fortran D has no such feature and more importantly, the underlying architecture in each processor of the multiprocessor configuration has been assumed to be scalar (see section 1.2), we have handled control dependences in a slightly different manner.

A control dependence edge (`deptype = IS_CONTROL` in section 5.4) is added to the DDG between every `IF` statement and the statements in its `THEN`/`ELSE` branches. The variables accessed in the evaluation of a condition are treated like any other use and can also participate in other data dependences. This is illustrated in Figure 5.2.

We observe from the DDG in Figure 5.2 that a control dependence edge between S_1 and S_2 has been introduced. These statements are also involved in an anti-dependence (use of `A(I+1)` in S_1 followed by a define in S_2) and a flow-dependence (define of `A(I+1)` in one iteration is followed by the use of `A(I)` in the next iteration).

In an `IF-THEN-ELSE` statement, the statements in the two branches can also

be involved in a dependence as long as they are not in the same iteration i.e., direction vector $(=, =, \ldots, =)$ is infeasible for two references in different branches of the same conditional statement. During all graph operations, a control dependence is treated on par with a data dependence. The restructuring algorithms work with little/no modification.

FRAMES is particularly severe on the use of GOTO statements within a loop, since the flow of control in different iterations becomes much more difficult to trace. Hence, the loops surrounding the GOTO are forcibly serialized unless any of the loops have been FORCE_PARALLELized by the user. This is a heavy penalty we are forcing the user to pay for using GOTOs within a loop—our humble contribution to the cause of structured programming.

5.8 The back end: techniques for improving parallelism detected

The BACK END1 mentioned in Figure 1.1 of Chapter 1 converts the IR back to a FORTRAN D program to enable the restructured program to be viewed by the user. In effect, it does the reverse function of the parsing stage. No sophisticated tools for interactive compilation have currently been developed as part of FRAMES. FRAMES dumps the DDG for those portions of the loops which could not be parallelized on to a log file. A user may analyze this file in conjunction with the input program and give the appropriate compiler directives for parallelizing/serializing the loops of interest. The compiler directives that are currently supported by FRAMES are FORCE_PARALLEL, FORCE_SERIAL and MAX_TRIPS (explained in Chapter 4).

Figure 5.3 gives a summary of the of the different stages in restructuring in FRAMES.

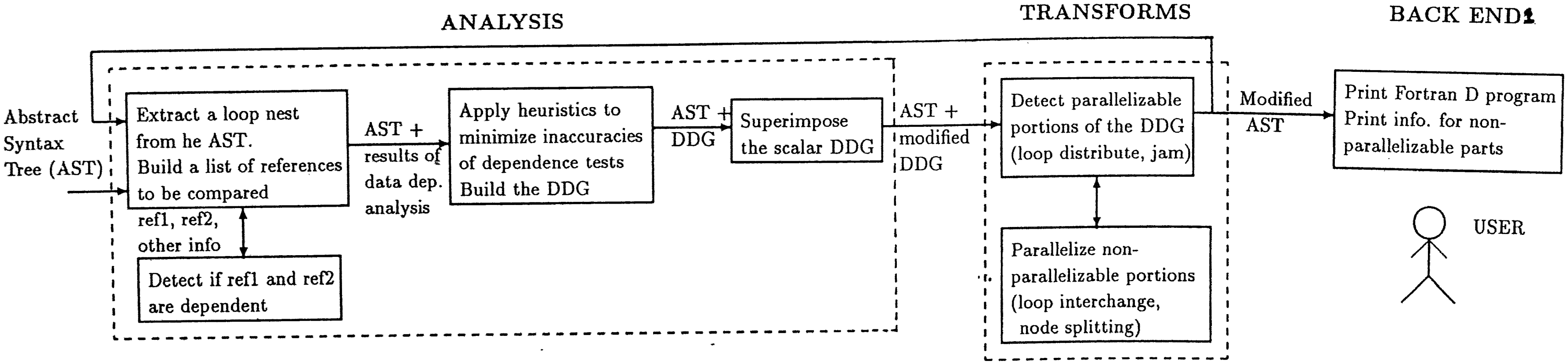


Figure 5.3: Different stages in restructuring in FRAMES

Chapter 6

Conclusions

To conclude with, we summarize the utility of FRAMES in the process of automated restructuring and review the progress made in this direction. The advantages of this aid have been mentioned earlier in Chapter 1. In this chapter, we present the results obtained when the existing prototype of FRAMES was run on some standard benchmarking routines. Suggestions for future work and the drawbacks of FRAMES are discussed.

Although FRAMES is being built for an MES model, it can easily be tailored to meet the requirements of some other model such as an MEA (Multiple Execution Array of vector processors). The statements in the innermost parallel loop may be converted to vector operations by having another phase after restructuring (refer Figure 1.1).

6.1 Results

The effect of any decisions taken during implementation manifest themselves in the results obtained. FRAMES has been tested extensively on two of the standard benchmarking packages, LINPACK and the Livermore kernels. In addition, we have also tested FRAMES with many programs drawn at random

from the user space of CONVEX *C-220* machine at IIT Kanpur. We include below the output generated by FRAMES for some programs.

6.1.1 Matrix multiplication

Matrix multiplication is an application that is well suited for parallelization. Since the dot products that are computed for each element of the product matrix are more or less independent of one another, a significant speedup can be obtained by parallelizing these computations. Matrix multiplication is a good example to demonstrate the power of loop interchanging.

Input program:

```
      dimension a(20,20), b(20,20), c(20,20)
c     perform matrix multiplication
      do 10 i=1,10
         do 20 j=1,10
            do 30 k=1,10
               c(i,j) = a(i,k)*b(k,j) + c(i,j)
30          continue
20       continue
10    continue
      stop
      end
```

FRAMES output:

```
      dimension a(20,20), b(20,20), c(20,20)
c     perform matrix multiplication
      do k=1,10
         forall i=1,10
            forall j=1,10
               c(i,j)=a(i,k)*b(k,j)+c(i,j)
            endfor
         endfor
      enddo
      stop
      end
```

Since the outermost loop is involved in the computation of the repeated sum of `c(i,j)`, it is an inherently sequential loop. Thus the parallelism exploited by FRAMES in this case, is the maximum.

6.1.2 The dmxpy routine (LINPACK)

The **dmxpy** routine in the LINPACK library of benchmarks essentially multiplies a matrix **m** by a vector **x** and adds the result to a vector **y**. The actual LINPACK program has been altered in that the matrices are declared to be of real type rather than double precision.

Input program:

```
c       benchmarking program dmxpy of LINPACK
        real y(1000), x(1000), m(1000, 1000)
c
c       purpose:
c       multiply matrix m times vector x and add the result to vector y.

c       cleanup odd vector
        j = mod(n2,2)
        if (j .ge. 1) then
          do 10 i = 1, n1
            y(i) = y(i) + x(j)*m(i,j)
10        continue
        endif
c
c       cleanup odd group of two vectors
        j = mod(n2,4)
        if (j .ge. 2) then
          do 20 i = 1, n1
            y(i) = y(i) + x(j-1)*m(i,j-1) + x(j)*m(i,j)
20        continue
        endif
c
c       cleanup odd group of four vectors
        j = mod(n2,8)
        if (j .ge. 4) then
          do 30 i = 1, n1
            y(i) = y(i)+x(j-3)*m(i,j-3)+x(j-2)*m(i,j-2)+x(j-1)*m(i,j-1)
     1             +x(j)*m(i,j)
30        continue
        endif
```

```
c
c       cleanup odd group of eight vectors
        j = mod(n2,16)
        if (j .ge. 8) then
          do 40 i = 1, n1
            y(i) = y(i)+x(j-7)*m(i,j-7)+x(j-6)*m(i,j-6)+x(j-5)*m(i,j-5)
     1             + x(j-4)*m(i,j-4) + x(j-3)*m(i,j-3) + x(j-2)*m(i,j-2)
     1             + x(j-1)*m(i,j-1) + x(j)*m(i,j)
40        continue
        endif
c
c       main loop - groups of sixteen vectors
        jmin = j+16
        do 60 j = jmin, n2, 16
          do 50 i = 1, n1
            y(i) = y(i) + x(j-15)*m(i,j-15) + x(j-14)*m(i,j-14)
     1             +x(j-13)*m(i,j-13)+x(j-12)*m(i,j-12)+x(j-11)*m(i,j-11)
     1             +x(j-10)*m(i,j-10)+x(j- 9)*m(i,j- 9)+x(j- 8)*m(i,j- 8)
     1             + x(j- 7)*m(i,j- 7) + x(j- 6)*m(i,j- 6)
     1             + x(j- 5)*m(i,j- 5) + x(j- 4)*m(i,j- 4)
     1             + x(j- 3)*m(i,j- 3) + x(j- 2)*m(i,j- 2)
     1             + x(j- 1)*m(i,j- 1) + x(j)   *m(i,j)
 50       continue
 60     continue
        end
```

FRAMES output:

```
c benchmarking program dmxpy of LINPACK
        real y(1000), x(1000), m(1000,1000)
c
c       purpose:
c       multiply matrix m times vector x and add the result to vector y.
c
c       cleanup odd vector
c
        j = mod(n2,2)
        if (j.ge.1) then
           forall i=1,n1
              y(i)=y(i)+x(j)*m(i,j)
           endfor
        endif
c
c       cleanup odd group of two vectors
        j=mod(n2,4)
        if (j.ge.2) then
           forall i=1,n1
              y(i)=y(i)+x(j-1)*m(i,j-1)+x(j)*m(i,j)
           endfor
        endif
```

```
c
c       cleanup odd group of four vectors
        j=mod(n2,8)
        if (j.ge.4) then
           do 30 i=1,n1
              y(i)=y(i)+x(j-3)*m(i,j-3)+x(j-2)*m(i,j-2)+x(j-1)*m(i,j-1)
     1        +x(j)*m(i,j)
           enddo
        endif
c
c       cleanup odd group of eight vectors
        j=mod(n2,16)
        if (j.ge.8) then
           forall i=1,n1
               y(i)=y(i)+x(j-7)*m(i,j-7)+x(j-6)*m(i,j-6)+x(j-5)*m(i,j-5)
     1         +x(j-4)*m(i,j-4)+x(j-3)*m(i,j-3)+x(j-2)*m(i,j-2)
     1         +x(j-1)*m(i,j-1) + x(j)*m(i,j)
           endfor
        endif
c
c       main loop - groups of sixteen vectors
        jmin=j+16
        do j=jmin,n2,16
           forall i=1,n1
              y(i)=y(i)+x(j-15)*m(i,j-15)+x(j-14)*m(i,j-14)
     1        +x(j-13)*m(i,j-13)+x(j-12)*m(i,j-12)+x(j-11)*m(i,j-11)
     1        +x(j-10)*m(i,j-10)+x(j-9)*m(i,j-9)+x(j-8)*m(i,j-8)
     1        +x(j-7)*m(i,j-7)+x(j-6)*m(i,j-6)
     1        +x(j-5)*m(i,j-5)+x(j-4)*m(i,j-4)
     1        +x(j-3)*m(i,j-3)+x(j-2)*m(i,j-2)
     1        +x(j-1)*m(i,j-1)+x(j)*m(i,j)
           endfor
        enddo
        end
```

We observe in this case that all the DO loops except the last nest have been parallelized. The last nest has been partially parallelized by loop interchanging. FRAMES has withheld parallelization of the loop j in this nest because a possible output-dependence has been sensed in the statement that computes y(i) in each iteration of j. Since the computation of y(i) uses the previous value of y(i), this loop too is inherently sequential.

6.1.3 A routine from Livermore kernel 13

The example program given below highlights the usefulness of loop distribution.

Input program :

```
c Benchmark program taken from Livermore kernel 13
        dim a(100,100), b(100,100), c(100,100), d(100,100)
        dim and(100,100), e(100), f(100), g(100), h(100)
        do 80 i = 1, n
            i1 = d(1,i)
            j1 = d(2,i)
            i1 = 1 + and(i1,63)
            j1 = 1 + and(j1,63)
            d(3,i) = d(3,i) + b(i1,j1)
            d(4,i) = d(4,i) + c(i1,j1)
            d(1,i) = d(1,i) + d(3,i)
            d(2,i) = d(2,i) + d(4,i)
            i2 = d(1,i)
            j2 = d(2,i)
            i2 = and(i2,63)
            j2 = and(j2,63)
            d(1,i) = d(1,i) + g(j2+32)
            d(2,i) = d(2,i) + h(j2+32)
            i2 = i2 + e(i2+32)
            j2 = j2 + f(j2+32)
            a(i2,j2) = a(i2,j2) + 1.0
80      continue
        stop
        end
```

FRAMES output :

```
c Benchmark program taken from Livermore kernel 13
        dim a(100,100), b(100,100), c(100,100), d(100,100)
        dim and(100, 100), e(100), f(100), g(100), h(100)
        forall i=1,n
           i1=d(1,i)
           i1=1+and(i1,63)
           j1=d(2,i)
           j1=1+and(j1,63)
           d(3,i)=d(3,i)+b(i1,j1)
           d(1,i)=d(1,i)+d(3,i)
           d(4,i)=d(4,i)+c(i1,j1)
           d(2,i)=d(2,i)+d(4,i)
        endfor
```

```
do i=1,n
   i2=d(1,i)
   j2=d(2,i)
   i2=and(i2,63)
   j2=and(j2,63)
   d(1,i)=d(1,i)+g(j2+32)
   d(2,i)=d(2,i)+h(j2+32)
   i2=i2+e(i2+32)
   j2=j2+f(j2+32)
   a(i2,j2)=a(i2,j2)+1.0
enddo
stop
end
```

A few remarks on the output generated by FRAMES for this example would be in order. The serial loop is an effect of the presence of `i2` and `j2` in the subscripts of array a. Since the bounds of `i2` and `j2` are not known, a conservative estimate will have to be made that there is an output-dependence in the computation of the elements of a. An improvement is possible if we promote the scalars `i2` and `j2` to vector references. Alternatively, the `FORCE_PARALLEL` directive can be applied if the user is convinced that these output-dependences are spurious. The following output from [9] has been obtained by promoting scalars to vectors.

```
c Better output for the Livermore kernel 13 example
        forall i = 1, n
            i1 = d(1,i)
            j1 = d(2,i)
            i1 = 1 + and(i1,63)
            j1 = 1 + and(j1,63)
            d(3,i) = d(3,i) + b(i1,j1)
            d(4,i) = d(4,i) + c(i1,j1)
            d(1,i) = d(1,i) + d(3,i)
            d(2,i) = d(2,i) + d(4,i)
            i2 = d(1,i)
            j2 = d(2,i)
            i2 = and(i2, 63)
            j2 = and(j2, 63)
            d(1,i) = d(1,i) + g(j2+32)
            d(2,i) = d(2,i) + h(j2+32)
            i2v(i) = i2 + e(i2+32)
            j2v(i) = j2 + f(j2+32)
        endfor
```

```
do i = 1, n
    i2 = i2v(i)
    j2 = j2v(i)
    a(i2,j2) = a(i2,j2) + 1.0
enddo
stop
end
```

6.1.4 An example of node splitting

This example program from [13] has been included to demonstrate the effect of node splitting.

Input program :

```
c       example from Polychronopoulos, pg. 23
        dim a(50), b(50), c(50), d(50)
        read(*,*) n
        do 10 i=1,n
           a(i) = b(i) + c(i)
           d(i) = a(i-1) * a(i+1)
10      continue
        stop
        end
```

FRAMES output:

```
c       example from Polychronopoulos, pg. 23
        dim a(50), b(50), c(50), d(50)
        dim frtmp1(50)
        read(*,*) n
        forall i=1,n
           frtmp1(i)=a(i+1)
        endfor
        forall i=1,n
           a(i)=b(i)+c(i)
        endfor
        forall i=1,n
           d(i)=a(i-1)*frtmp1(i)
        endfor
        stop
        end
```

The anti-dependence due to the references to a(i) and a(i+1) in the two assignment statements has been broken in the restructured program enabling

parallelization. `frtmp1` is the variable introduced by FRAMES during node splitting.

6.2 Limitations of FRAMES: suggestions for future work

Notwithstanding the parallelism tapped at present by FRAMES, a lot more can be done to improve the quality of the output. The results demonstrated an example of how non-promotion of scalars inhibited the parallelism detected. We enumerate below the current limitations of FRAMES and suggest areas in which more work could be done.

- Since distance vectors cannot be found out by the inexact methods currently used by FRAMES, transformations such as loop skewing or cycle shrinking cannot be applied. For example, the loop in the Jacobi problem reproduced below cannot be parallelized currently by FRAMES.

```
DO I = 2, N-1
   DO J = 2, N-1
      A(I, J) = (A(I+1, J) + A(I-1, J) + A(I, J+1) +
                   A(I, J-1))/4
   ENDDO
ENDDO
```

  Another benefit of storing the distance of a dependence is that direction vectors may be compressed. (Inaccuracies caused by compression can be filtered out by examining the dependence distances.)

- The inability to promote scalar references to vectors and array references to a higher dimension effectively retains many of "false" dependences.

- While the directives for FRAMES are useful, they are too coarse-grained. A user is better equipped to handle queries such as 'does this dependence

exist between reference x and y?'. It is imperative that directives be added through which a user may allow certain dependences to be neglected.

- Procedures and functions introduce many subtleties since the procedure/ function body may introduce additional dependences if the call site is within a loop.

Besides these, a more robust parser that handles all the features of Fortran D, data distribution and the issue of data posting mentioned in Chapter 5 need to be examined. Many of these are being addressed in [16].

Appendix A

Summary of Fortran D

The chief drawback of most high-level languages used for programming multiprocessors is that they are either too difficult to use (such as message-passing Fortran 77) or too inflexible for so complex an architecture (such as Fortran 90). Most important of all, these languages pay no regard to the problem of data distribution/data decomposition. Fortran D is a superset of Fortran 77, the language most extensively used in scientific applications and supports a rich variety of statements for data decomposition. We provide here a summary of the main features of Fortran D that are not a part of Fortran 77. Refer [Fox] for a thorough report on Fortran D.

The `FORALL` parallel construct: The `FORALL` construct has the same syntax as that of the `DO` construct in Fortran. All iterations of a `FORALL` loop may be executed in *any* order in parallel and there is no communication between the processors executing different iterations. Thus execution of the different loop iterations in a `FORALL` is non-deterministic. The values used during execution of a `FORALL` are either those defined before the loop was entered or those defined in that iteration of the `FORALL`.

Data decompositions and distributions

Data parallel applications have two levels of parallelism. The structure of the underlying computation induces a *problem mapping* i.e., the alignment of arrays with respect to one another, both within and across array dimensions. The problem mapping is independent of machine considerations. Moreover, these arrays need to be distributed on to the actual machine (*machine mapping*), keeping in mind the finite resources that are offered by the machine.

A *data decomposition* is an abstract problem or index domain—it is a desired view of an array. A `DECOMPOSITION` statement declares the name, dimensionality and size of a decomposition for later use. For instance,

```
DECOMPOSITION A(N), B(N,N)
```

defines `A` to be a one-dimensional decomposition of size `N` and `B` to be a two-dimensional decomposition.

Data alignment

An array used in a program should be aligned with a particular decomposition. For example, if a two-dimensional array is to be mapped onto a decomposition that is a transpose of the array being mapped, we may specify

```
REAL X(N,N)
DECOMPOSITION A(N,N)
ALIGN X(I,I) with A(J,I)
```

`I` and `J` used in the subscripts of `X` are called as placeholders. In effect, the first and second dimensions of `X` have been respectively mapped to the second and first dimensions of decomposition `A`.

In general, the alignment specified can be either intra-dimensional or inter-dimensional to get any desired mapping.

Data distribution

Arrays are aligned with a decomposition as part of the problem mapping. But decompositions themselves need to be mapped onto the underlying machine. The `DISTRIBUTE` statement has an attribute called the type of decomposition. Since the `DISTRIBUTE` statement would decide the processors' memories into which a decomposition is to be distributed, selecting the right attribute is very critical. The commonly used attributes are

1. `BLOCK`– divide the decomposition into contiguous chunks of size N/P if there are N elements and P processors in the decomposition. Each processor gets a block.
2. `CYCLIC`– divide the elements among the N processors in a round-robin fashion. Thus every processor has the Pth element in the decomposition.
3. `BLOCK_CYCLIC`– divide into contiguous chunks of size M and then assign these chunks in a round-robin fashion similar to `CYCLIC`.

In FRAMES, we have made the simplistic assumption that every array used in the program is aligned with a decomposition that is a mirror image of itself.

Reductions

A `REDUCE` statement is an operation on a collection of data that results in new data of a lesser dimensionality, usually a scalar value. For example,

```
REDUCE(SUM, S, X(I))
```

applies the function SUM on the elements X(I) and puts the result in S.

Appendix B

Grammar of the language accepted by the parser

As mentioned before, the parser that is currently part of FRAMES does not accept the complete language of Fortran D but only a subset of it. At the current stage of development, we have taken care to include all features of the language that would directly affect the exploitation of parallelism. Among the Fortran D features that are *not* supported are

- labeled `COMMON` statement
- all data types barring `REAL` and `INTEGER`
- different types of distributions (see Appendix A)

We give here the grammar of the language accepted by the parser. Nuances such as disambiguating function calls and array references, etc. are handled by the action routines for each grammar rule.

```
program          : declarations commands END
                 | declarations commands I_CONS END
                 | declarations commands END procedures
                 | declarations commands I_CONS END procedures
                 ;

declarations     : declarations DIM array_id_list
                 | declarations COMMON id_list
                 | declarations EQUIVALENCE equiv_list
                 | declarations INTEGER mixed_id_list
                 | declarations REAL mixed_id_list
                 | declarations DISTRIBUTE array_id_list
                 |
                 ;

array_id_list    : array_id_list ',' ID '(' fixed_bounds_list ')'
                 | ID '(' fixed_bounds_list ')'
                 ;

fixed_bounds_list : fixed_bounds_list ',' I_CONS
                 | I_CONS
                 ;

id_list          : id_list ',' ID
                 | ID
                 ;

mixed_id_list    : mixed_id_list ',' ID
                 | mixed_id_list ',' ID '(' fixed_bounds_list ')'
                 | ID
                 | ID '(' fixed_bounds_list ')'
                 ;

read_list        : ID ',' read_list
                 | ID '(' bounds_list ')' ',' read_list
                 | ID
                 | ID '(' bounds_list ')'
                 ;

bounds_list      : ID ',' bounds_list
```

```
                   | I_CONS ',' bounds_list
                   | ID
                   | I_CONS
                   ;

equiv_list         : '(' id_list ')'
                   | equiv_list ',' '(' id_list ')'
                   ;

commands           : commands statement
                   | commands I_CONS statement
                   | I_CONS statement
                   | statement
                   ;

statement          : assign_stat
                   | STOP
                   | READ read_list
                   | WRITE write_list
                   | FORMAT
                   | GOTO I_CONS
                   | CALL ID '(' call_parm_list ')'
                   | IF '(' condition ')' statement
                   | IF '(' condition ')' THEN if_body
                   | loop_directive DO opt_label ID '=' expr ',' expr
                       loop_incr do_body
                   | FORALL ID '=' expr ',' expr
                       loop_incr forall_body
                   ;

if_body            : commands ENDIF
                   | commands ELSE commands ENDIF
                   ;

loop_directive     : loop_directive DIRECTIVE FORCE_PARALLEL
                   | loop_directive DIRECTIVE MAX_TRIPS I_CONS
                   | loop_directive DIRECTIVE NO_PARALLEL
                   ;

loop_incr          : ',' expr
```

```
                |
                ;

assign_stat     : ID '=' expr
                | ID '(' array_indices ')' '=' expr
                ;

expr            : '-' multerm
                | '+' multerm
                | multerm
                | expr '+' multerm
                | expr '-' multerm
                ;

multerm         : multerm '*' factor
                | multerm '/' factor
                | factor
                ;

factor          : factor EXP factor
                | ID
                | ID '(' array_indices ')'
                | I_CONS
                | R_CONS
                | '(' expr ')'
                ;

array_indices   : expr ',' array_indices
                | expr
                ;

write_list      : ID ',' write_list
                | ID '(' bounds_list ')' ',' write_list
                | QUOTED_STRING ',' write_list
                | ID '(' bounds_list ')'
                | QUOTED_STRING
                | ID
                ;

call_parm_list  : expr ',' call_parm_list
```

```
                | expr
                |
                ;

condition       : rel_expr
                ;

rel_expr        : '(' rel_expr ')'
                | NOT rel_expr
                | rel_expr AND rel_expr
                | rel_expr OR rel_expr
                | expr EQ expr
                | expr NE expr
                | expr GT expr
                | expr GE expr
                | expr LT expr
                | expr LE expr
                ;

do_body         : commands opt_label iterator_end
                ;
forall_body     : commands END_FOR
                ;

opt_label       : I_CONS
                |
                ;

iterator_end    : CONTINUE
                | ENDDO
                ;

procedures      : subprogram
                | procedures subprogram
                ;

subprogram      : function
                | subroutine
                ;
```

function	: *func_type* `FUNCTION` `ID` '(' *parm_list* ')' *declarations* *function_body*
	;
func_type	: `REAL`
	| `INTEGER`
	;
parm_list	: `ID`
	| *parm_list* ',' `ID`
	;
function_body	: *fn_body* `END`
	;
fn_body	: *commands* `RETURN` *fn_body*
	| *commands* `I_CONS` `RETURN` *fn_body*
	| `RETURN`
	| `I_CONS` `RETURN`
	|
	;
subroutine	: `SUBROUTINE` `ID` '(' *parm_list* ')' *declarations* *procedure_body*
	;
procedure_body	: *function_body*
	;

Bibliography

[1] Aho, A. V., Sethi, R. and Ullman, J. D., *Compilers: Principles, Techniques, and Tools.*, Addison-Wesley, 1986.

[2] Allen, R. and Kennedy, K., *Automatic Translation of Fortran Programs to Vector Form*, ACM Transactions on Programming Languages and Systems, 9(4), Oct. 1987, pp. 491-542.

[3] Banerjee, U., *Dependence Analysis for Supercomputers*, Kluwer Academic, 1988.

[4] Cytron, R. and Burke, M., *Interprocedural Analysis and Parallelization*. Proceedings of the SIGPLAN '86 Symposium on Compiler Construction, 21(7), Jul. 1986, pp. 162-175.

[5] Fox, G., Hiranandani, S., Kennedy, K., Koebel, C., Kremer, U., Tseng, C. and Wu, M., *Fortran D Language Specification*, Tech. Rep. TR 90-141, Dept. of Computer Science, Rice University, Dec. 1990.

[6] Gautam, S., *Fortran D parallelizing compiler: Scheduling phase*, Dept. of CSE, IIT Kanpur, Jan. 1993.

[7] Hiranandani, S., Kennedy, K. and Tseng, C., *Compiling* Fortran D *for MIMD Distributed Memory Machines*, Commun. ACM, 35(8), Aug. 1992, pp. 66-80.

[8] Kong, X., Klappholz, D. and Psarris, K., *The I Test : An Improved Dependence Test for Automatic Parallelization and Vectorization*, IEEE Trans. Para. and Distrib. Syst., 2(3), July 1991, pp. 342-349.

[9] Levesque, J. M. and Williamson, J. W., *A Guidebook to Fortran on Supercomputers*, Academic Press Inc., 1989.

[10] Maydan, D. E., Hennessy, J. L. and Lam, M. S., *Efficient and Exact Data Dependence Analysis*, Proceedings of the ACM SIGPLAN '91 Conference

on Programming Language Design and Implementation, June 1991, pp. 1-14.

[11] Nori, K. V., Kumar, S. and Kumar, M. P., *Retrospective of the PQCC Compiler Structure*, Proceedings of Foundations of Software Technology and Theoretical Computer Science, Dec. 1987, LNCS vol. 287, Springer-Verlag, pp. 500-527.

[12] Padua, D. A., *Advanced Compiler Optimizations for Supercomputers*, Commun. ACM, 29(12), Dec. 1986, pp. 1184-1200.

[13] Polychronopoulos, C. D., *Parallel Compilers*, Kluwer Academic Publishers, 1987.

[14] Pugh, W., *A Practical Algorithm for Exact Array Dependence Analysis*. Commun. ACM, 35(8) Aug. 1992), pp. 102-114.

[15] Singh, R. and Purohit, V. D., *A Tool for Vectorizing Sequential Programs*, BTech project report, Dept. of CSE, IIT Kanpur, Apr. 1991.

[16] Singh, R., MTech thesis report (to be published), Dept. of CSE, IIT Kanpur, 1993.

[17] Tarjan, R. E., *Depth first search and linear graph algorithms*, SIAM J. Comput., 1(2) 1972, pp.146-160.

[18] Wolf, M. and Lam, M. S. *A Loop Transformation Theory and an Algorithm to Maximize Parallelism*, IEEE Trans. on Para. and Distrib. Syst., 2(4) Oct. 1991, pp. 452-471.

[19] Wolfe, M. J. *Optimizing Supercompilers for Supercomputers*, Ph.D. Thesis, University of Illinois at Urbana-Champaign, 1982.

[20] Wolfe, M. *Advanced Loop Interchanging*, Proceedings of the 1986 International Conference on Parallel Processing, pp. 536-543.

[21] Wolfe, M. *Loop Skewing: The Wavefront Method Revisited*, Int. J. of Para. Prog., 15(4), 1986, pp. 279-293.

[22] Wu, T. and Lewis, T. G. *Parallelizing While Loops*, International Conference on Parallel Processing (II), 1990, pp. 1-8.

[23] *CONVEX FORTRAN Optimization Guide*, CONVEX Computer Corporation, 1990.